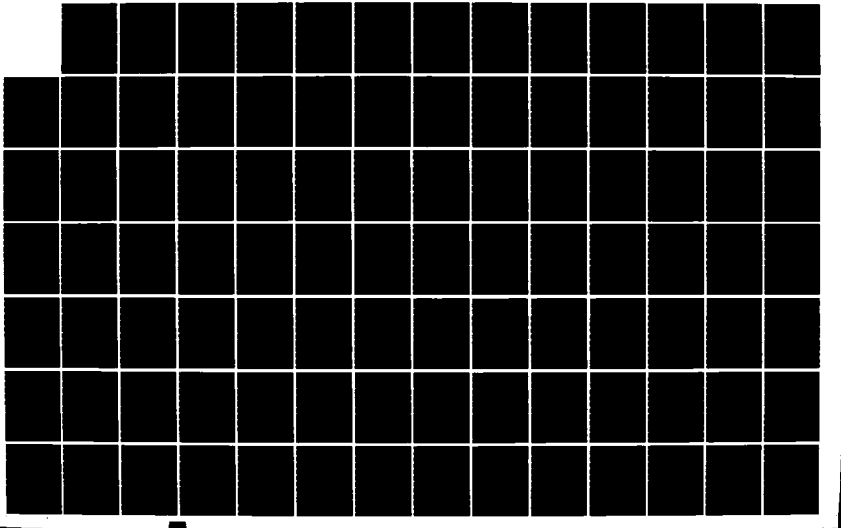
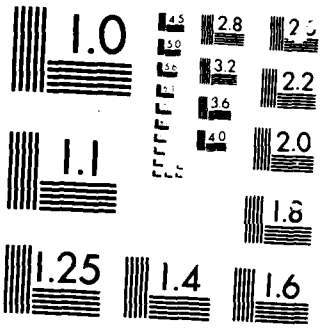
AD-A166 287 DESIGN OF A DATA STORAGE HIERARCHY DSH-III--SOFTWARE & HARDWARE(U) ALFRED P SLOAN SCHOOL OF MANAGEMENT CAMBRIDGE MA M J ABRAHAM MAR 86 M010-8603-19 1/2

UNCLASSIFIED N00039-83-C-0463 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART

DESIGN OF A DATA STORAGE HIERARCHY: DSH-III -- SOFTWARE & HARDWARE

Michael J. Abraham

Technical Report #19

March 1986

Center for Information Systems Research

Massachusetts Institute of Technology
Sloan School of Management
77 Massachusetts Avenue
Cambridge, Massachusetts 02139

Contract Number N00039-83-C-0463

Internal Report Number M010-8603-19

(12) 4

DESIGN OF A DATA STORAGE HIERARCHY: DSH-III -- SOFTWARE & HARDWARE

Michael J. Abraham

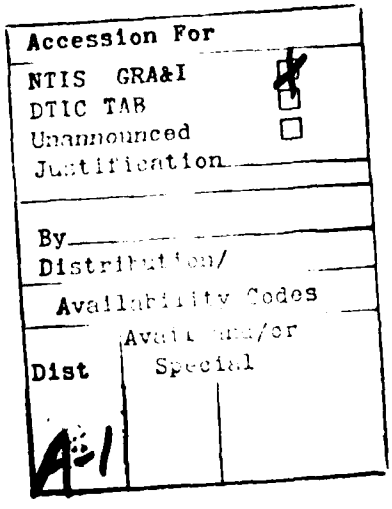
Technical Report #19

March 1986

Principal Investigator:
Professor S. E. Madnick

Prepared for:
Naval Electronics Systems Command
Washington, D.C.

Table of Contents



Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Technical Report #19	2. GOVT ACCESSION NO. AD-A166 287	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Design of a Data Storage Hierarchy: DSH-III -- Software & Hardware		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER M010-8603-19
7. AUTHOR(s) Michael J. Abraham		8. CONTRACT OR GRANT NUMBER(s) N00039-83-C-0463
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center for Information Systems Research Sloan School of Management, M.I.T. Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE March 1986
		13. NUMBER OF PAGES 122
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Hierarchical decomposition and distributed control, storage hierarchy, user interface, demand paging, data location, overflow handling, READ & WRITE strategies, READ & WRITE algorithms

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Modern organizations are becoming increasingly reliant on the storage and processing of very large data bases in support of their accounting, operational control, and high-level decision-making functions. Contemporary Data Base Management Systems (DBMS's) are capable of handling data bases on the order of a trillion (10^{12}) bits of data, and can process transactions at rates of up to 100 queries per second.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

However, it is expected that future high-performance DBM's will be required to provide storage capacities and transaction rates several orders of magnitude greater than those of any current systems. It is not unreasonable to project requirements of a quadrillion (10^{15}) bits and one million database accesses per second.

As DBMS's become ever more integral parts of many organizations' operations the costs of system failure or unavailability increase correspondingly. Accordingly, there is a growing need for "fault tolerant" systems which can provide continuous availability in the presence of many types of internal component failure (both hardware and software).

To meet the requirements for increased speed, capacity, and availability the IMS Data Base Computer (INFOPLEX) employs a highly parallel, distributed control architecture. The preliminary INFOPLEX design consists of two logically and physically separate components: a physical storage hierarchy which consists of a series of micro-processor controlled storage devices functioning as a very large (10^{15} bits) virtual memory; and a functional hierarchy, consisting of a series of micro-processor clusters, which provide user interfaces, security, and memory management facilities for an INFOPLEX database system. The general structure of an INFOPLEX data storage hierarchy has been developed and a preliminary set of control algorithms has been proposed.

This report presents a refined set of control algorithms which take advantage of theoretical properties of INFOPLEX-like storage systems. Section 2 presents an overview of the INFOPLEX design issues and tradeoffs related to performance and reliability, and develops a general strategy for efficient READ/WRITE control and reliable fault handling. Section 3 develops the functional design of DSH-III in more detail, including functional descriptions of the system components and their interfaces. Sections 4 and 5 detail the algorithms and protocols supporting the READ and WRITE operations, respectively. Finally, Section 6 summarizes this report, and indicates the directions for future research.

1 INTRODUCTION

Modern organizations are becoming increasingly reliant on the storage and processing of very large data bases in support of their accounting, operational control, and high-level decision-making functions. Contemporary Data Base Management Systems (DBMS's) are capable of handling data bases on the order of a trillion (10^{12}) bits of data [26], and can process transactions at rates of up to 100 queries per second [2]. However, it is expected that future high-performance DBMS's will be required to provide storage capacities and transacton rates several orders of magnitude greater than those of any current systems. It is not unreasonable to project requirements of a quadrillion (10^{15}) bits and one million database accesses per second [18].

As DBMS's become ever more integral parts of many organizations' operations the costs of system failure or unavailability increase correspondingly. Accordingly, there is a growing need for "fault tolerant" systems which can provide continuous availability in the presence of many types of internal component failure (both hardware and software).

To meet the requirements for increased speed, capacity, and availability the IMS Data Base Computer (INFOPLEX) employs a highly parallel, distributed control architecture. The preliminary INFOPLEX design consists of two logically and physically separate components: a physical storage hierarchy which consists of a series of micro-processor controlled storage devices functioning as a very large (10^{15} bits) virtual memory; and a functional hierarchy, consisting of a series of micro-processor clusters, which provide user interfaces,

security, and memory management facilities for an INFOPLEX database system [11]. The general structure of an INFOPLEX data storage hierarchy has been developed and a preliminary set of control algorithms has been proposed [15].

This report presents a refined set of control algorithms which take advantage of theoretical properties of INFOPLEX-like storage systems [16, 3]. Section 2 presents an overview of the INFOPLEX design issues and tradeoffs related to performance and reliability, and develops a general strategy for efficient READ/WRITE control and reliable fault handling. Section 3 develops the functional design of DSH-III in more detail, including functional descriptions of the system components and their interfaces. Sections 4 and 5 detail the algorithms and protocols supporting the READ and WRITE operations, respectively. Finally, Section 6 summarizes this report, and indicates the directions for further research.

2 OVERVIEW OF DSH-III DESIGN

2.1 Introduction

The INFOPLEX Data Storage Hierarchy III (DSH-III) is a model for a very large, high-speed, reliable storage system. The primary design objective of DSH-III is to provide a user (in particular the INFOPLEX Functional Hierarchy) with a very large, high-speed virtual address space. As will be seen, DSH-III takes complete responsibility for all physical data management, control of the various storage devices in the system, and recovery from almost all types of single-component failures. By incorporating the intelligence needed to perform these functions into the storage system, DSH-III is able to provide a user with a very simple, clean, easy to use interface. In particular, DSH-III can provide almost complete memory system functionality to a user through two primitive operations - READ and WRITE. This should be contrasted with the more than thirty Channel Commands needed to fully utilize an IBM 33xx series disk [14].

While this paper presents the details of only two primitive operations - READ and WRITE - additional primitives can be added as experience indicates in order to increase the usability and flexibility of DSH-III. A partial list of such primitives might include:

TEST_AND_SET - provides an atomic conditional update operation which can be be used to support P and V [8] synchronization operations;

SET_SECURE - allows a user to select a portion of the DSH-III virtual address space for special high-reliability handling, such as automatic

replication and duplication of data. This facility might be used to protect data of an especially critical nature, such as system control tables. The Tandem [4] computer is an example of a system which uses automatic duplication to enhance availability;

BLOCK_MOVE - allows a user to transfer large data blocks from one location in virtual memory to another. By simply modifying the mapping of real to virtual memory, this operation could be accomplished without any actual data movement;

BLOCK_ZERO - allows a user to initialize (set to zeroes) a large area of virtual memory.

2.2 Basis of DSH-III Design

The fundamental rationale for the design of DSH-III to be presented in this paper is based on three principles: 1) employing a range of storage technologies, 2) taking advantage of locality of reference, and 3) hierarchical decomposition and distributed control.

2.2.1 Range of Storage Technologies

A basic problem which constrains the design of any high-speed, high-capacity storage system is that no single storage technology can meet the requirements for both speed and capacity within reasonable cost constraints. For example, high-speed semi-conductor RAM can support random access times of 50ns but costs on the order of $1.00 per

byte. At the other end of the cost/capacity/speed spectrum are mass storage devices such as automated tape handlers, which can store large quantities of data at a cost of only .0005 cents per byte, but which have access times up to seven orders of magnitude slower than high-speed RAM. In between these two extremes are a range of storage device technologies as shown in Table 2.1.

Storage Level	Example	Random Access Time	Sequential Transfer Rate (bytes/sec)	Unit Capacity (bytes)	Unit Price (cents/byte)
1. Cache	HMOS RAM	50 ns	100M	32K	100
2. Main	NMOS RAM	1 us	16M	512K	10
3. Block	Magnetic Bubble Memory	100 us	8M	2M	2
4. Backing	High-Speed Drums	2 ms	2M	10M	0.5
5. Secondary	Disks	25 ms	1M	100M	0.02
6. Mass	Automated Tape Handlers	1 sec	1M	100B	0.0005

Table 2.1 - Summary of Storage Technologies

The approach taken in the design of the INFOPLEX Data Storage Hierarchy is to utilize a range of storage technologies, with the bulk of the data stored on inexpensive but relatively slow devices and automatically migrated to higher speed devices when it is accessed. This approach is logically equivalent to that used by cache based computer systems such as the IBM 3033 [12] and by mass storage systems such as the IBM 3850 [13].

2.2.2 Locality of Reference

In order for a multi-media storage system to take full advantage of its higher speed storage devices, it is desirable that the higher speed devices be accessed relatively more often than the lower speed devices. For this goal to be attainable, it is necessary that the database be subject to a non-homogeneous reference pattern. This non-homogeneity can be spatial or temporal and any database system which has this property is said to exhibit locality of reference [17]. Spatial locality refers to access patterns for which reference to any particular data item increases the probability that related data items will also be accessed. There are many examples of this phenomenon:

- sequential flow of control in a software module implies that reference to an instruction in program storage presages references to following instructions.
- reference to a particular field in a record stored in a file system is usually accompanied by references to other fields in the same record (or the same field in related records).

Temporal locality refers to access patterns for which consecutive references to data items are correlated in time. Automatic teller systems provide a typical example of this phenomenon. A common usage consists of a balance inquiry followed by a cash withdrawal, resulting in two accesses to the account balance within a short period of time. (In fact, the withdrawal alone, or any other database update, exhibits temporal locality due to the need to read the data item to be updated before writing the modified version of the data.)

INFOPLEX takes advantage of the fact that database systems do exhibit locality [25, 24], and that locality can be used to increase the relative utilization of the higher speed storage devices in the system.

There are three strategies for taking advantage of locality:

- static, where high-usage data, such as key system tables, is permanently allocated to higher speed devices
- manual, where it is the responsibility of the programmer to move data to higher speed devices when it is needed
- and automatic, where the system automatically migrates high usage data to fast devices and low usage data to slow devices.

One design strategy of INFOPLEX is to use automatic migration to take advantage of locality. This technique has the following advantages:

- it allows dynamic response to changing application loads and time-varying database content
- it relieves the programmer or system designer of the burden of allocating the data.

This strategy is analogous to that used by Multics which automatically migrates pages of virtual memory between high speed paging devices (e.g., drums) and lower speed devices (e.g., disks) in response to changes in the working sets of the active tasks in the system. [9] The precise manner in which automatic migration is implemented in DSH-III is described in Section 2.3.

2.2.3 Hierarchical Decomposition and Distributed Control

INFOPLEX organizes its heterogeneous array of storage devices using the principle of hierarchical decomposition and distributed control. The use of hierarchical decomposition leads to a conceptual system design such as that shown in Figure 2.1. This structure has been shown [19] to represent an efficient and effective method for integrating heterogeneous storage devices into a single system. There are three primary advantages to this structure.

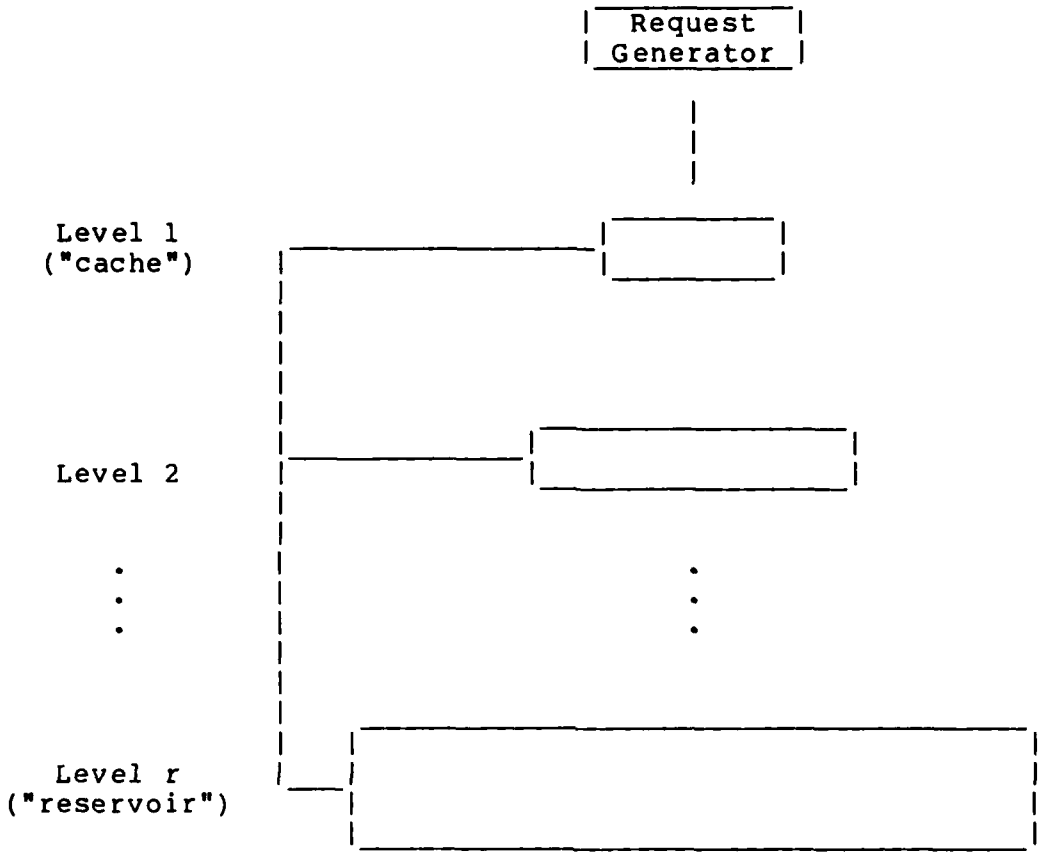


Figure 2.1 - Logical Structure of a Storage Hierarchy

First, the hierarchical structure supports the types of direct interlevel data transfer which are used by read, write, and automatic migration algorithms. This avoids the overhead associated with the indirect data path (i.e., drum to main memory to disk) used by the page migration scheme of Multics mentioned above.

Support for direct inter-level data transfer is a reflection of the second advantage of the hierarchical structure, namely that it facilitates the utilization of a distributed control strategy for the system. By this we mean that the basic system control and interlevel communication functions will be performed by micro-processor clusters

within each level. This strategy improves system performance by facilitating parallel and asynchronous operation within the hierarchy, as well as eliminating the potential reliability exposure that would be associated with a single controlling processor cluster.

Third, the design is inherently modular. This has four principal advantages

- the structure allows the use of common algorithms and software modules at each of the levels. This facilitates software design, especially in the area of interlevel communications protocols.
- if a level fails, the remaining levels form a system which is logically equivalent to the original (unfailed) system. This has important implications for the ability of the system to continue operation in the presence of failures.
- the modular structure facilitates the incorporation of new storage technologies into an INFOPLEX system. Thus the basic design should be relatively insensitive to the rapidly changing technology in this area.
- the structure allows the building of storage hierarchies with the number of levels and the types of storage device at each level customized to a particular application.

2.3 Design of an INFOPLEX Storage Hierarchy

This section presents the cost/performance/complexity/reliability tradeoffs and other issues underlying the design of DSH-III. We begin, in Section 2.3.1, with a general overview of the system topology implied by the discussion in Section 2.2. Next, Section 2.3.2 describes the interface between DSH-III and a user. Section 2.3.3 presents a justification for the data management strategies used in DSH-III. Based on these strategies, a specification for the READ algorithms used by DSH-III is developed in Section 2.3.4. Finally, the basic design of DSH-III is completed by the specification of WRITE algorithms for DSH-III in Section 2.3.5.

2.3.1 Overview of System Topology

Following the reasoning presented in the preceding sections leads to a conceptual system design such as the one shown in Figure 2.2.

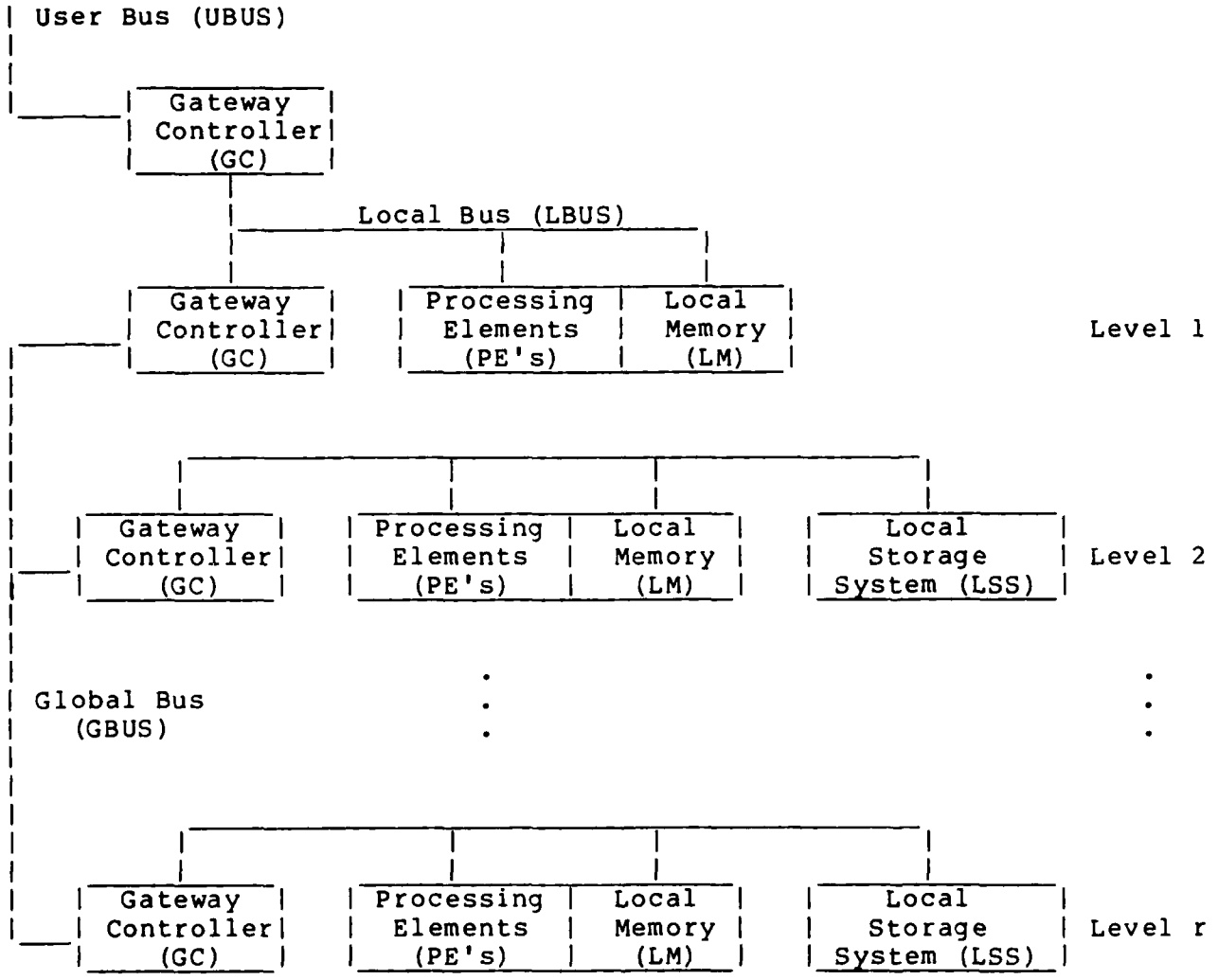


Figure 2.2 - Conceptual Structure of a Storage Hierarchy

The system consists of r storage levels, Level 1 to Level r, with Level 1 containing high-speed cache memory and Level r (the "reservoir") containing mass storage devices which contain a copy of

all the data in the database. One of the modules shown in Levels 2 to r is a Local Storage System (LSS), which consists of the physical devices holding the DSH-III database. LSS technology will vary from level to level, with higher levels using faster, but lower capacity, devices. In particular, because of the high response time requirements for the highest level, Level 1 will use the same storage technology for both Local Memory (LM) and LSS. For this reason, no separate LSS is shown for Level 1 in Figure 2.2.

All interlevel communication is performed via the Global Bus (GBUS). The major disadvantage of using a single GBUS instead of an inter-level bus between each adjacent pair of levels is that the parallelism of the system is reduced and thus bus contention and resultant queuing delays are increased. That this is an important consideration is shown by simulation studies of a storage hierarchy [15] which showed GBUS utilizations of over 80%. While 80% utilization of the GBUS has little impact on system performance (because the GBUS has a high bandwidth relative to other components of the system) it is clear that the GBUS could easily become a bottleneck under slightly different assumptions for component speeds or transaction loads. On the other hand, the logical GBUS could be implemented as multiple physical buses. In this case, adding extra capacity to a bottlenecked GBUS would not involve any great difficulty, but could be accomplished by replicating existing hardware structures. The potential disadvantage of using a single logical GBUS is clearly outweighed by the advantages offered by this structure. These advantages include

- the ability for a level to communicate with any other level, thus supporting "broadcasting" of information. As will be seen in Section 2.3.3, broadcasting greatly improves the efficiency of data movement algorithms and

eases system control problems by providing a means of synchronizing operations at different levels.

- increased availability, since the system structure facilitates "graceful degradation" and continued operation in the presence of level failures.

- more cost effective utilization of resources since the resource (the GBUS) is shared by all users (the levels) in the system. This facilitates the matching of bus capacity and demand.

Each storage level consistes of a number of storage devices and processing modules, interconnected via a Local Bus (LBUS). The interface between each level and the GBUS is provided by a Gateway Controller (GC) at each level. From the viewpoint of a GC, each level appears as an identical black box, i.e., the number and types of processors and storage devices within each level are transparent to the GC at that level. This conforms wi ı the concept of modularity and commonality of algorithms and software discussed in Section 2.2.3 above.

A Pended Bus Protocol [28] will be used for all buses in order to support the large number of devices on each bus.

Section 3 contains a more complete discussion of a possible hardware implementation of this hierarchical structure.

2.3.2 User Interface

Level 1 (the "cache" level) of the storage hierarchy serves as the interfa e between the user (the lowest level of the INFOPLEX Functional Hierarchy) and DSH-III. In particular this implies that Level 1 represents a shared cache structure, rather than a per user processor cache structure. Detailed simulation studies show that this single-bus, shared cache structure is a very effective and efficient

topology for providing high-speed, parallel, multi-processor access to the storage hierarchy. This structure, which is made feasible by the use of the Pended Bus protocol, has the significant advantage of greatly simplifying the cache consistency control problem. For a complete discussion of possible alternate topologies and the trade-offs among performance and consistency control policies the reader is referred to [1].

In addition to the usual GC at Level 1, there is another Gateway Controller which serves as an interface between the Level 1 LBUS and a User Bus (UBUS) which connects DSH-III and the Functional Hierarchy.

From the point of view of a user, DSH-III appears as a very large linear address space which is accessed via simple primitives such as

READ(request_id,virtual_address) and

WRITE(request_id,virtual_address,data).

As noted previously, the interface has been kept as simple as possible, and a user is completely isolated from the details of data management and error recovery. The intelligence to perform these functions is distributed throughout DSH-III.

The READ and WRITE operations are not atomic operations. This means that control is returned to a user after he issues one of these commands, but before the operation has completed. When the operation finally completes, the user is notified. This implies that a user may have multiple operations active simultaneously. Because of this, it is important to define exactly what results DSH-III will produce if READs and WRITEs are overlapped.

The simplest way to look at this problem is to think of WRITE as an atomic operation. If an operation starts or completes after a WRITE has been issued but before it has completed, its sequencing with respect to the WRITE is undefined. In other words, in the sequences

READ, WRITE, READ complete, WRITE complete, or

WRITE, READ, WRITE complete, READ complete

the READ may be considered to have come either before or after the WRITE. Therefore, the results of the READ may or may not reflect the results of the WRITE. The only guaranteed way to ensure that a READ will reflect the results of a WRITE is to issue the READ after the WRITE has completed.

The remainder of this chapter is devoted to explaining and justifying the data movement strategies used by DSH-III, and Sections IV and V present the details of the algorithms used by DSH-III to support the READ and WRITE primitives, respectively.

2.3.3 Data Movement Strategies

The objectives of the data movement strategies used by DSH-III are three-fold. First, the strategies attempt to take advantage of locality by migrating high usage data to the higher speed storage devices. Second, the strategies attempt to minimize unnecessary data movement within the system in order to reduce bus contention. Third, redundant copies of data blocks are maintained at various levels. This has the effect of increasing system reliability while greatly simplifying page migration algorithms.

The basic design decisions underlying the data movement strategies of DSH-III are

- when should a block of data be moved from a lower level to a higher level of the hierarchy?
- if the transfer of a data block to a higher level necessitates the removal of a block already in the higher level, how should the block to be removed be selected, and what should be done with it (e.g., should it be discarded or transferred to some other level)?
- how big should the basic unit of access and transfer (the "page size") be; should it be the same at all levels or differ from level to level?

2.3.3.1 Demand Paging with Replacement

The data retrieval algorithms of DSH-III are based on a demand fetch policy. Under this policy, a data block is moved from a lower to a higher level in the hierarchy only in response to an explicit READ request by a user. In other words, there is no attempt made to anticipate future retrieval requests (based on known usage patterns or locality considerations) by pre-fetching data blocks before they are explicitly requested. This does not mean that the system does not take advantage of spatial locality. As will be seen, DSH-III blocks data into pages of various sizes, and this policy does result in anticipatory retrieval of data stored in the same page as the data being explicitly retrieved. The point here is that no anticipatory fetches are made, even though some data is retrieved in anticipation of future use by fetches that were going to be performed anyway in response to an explicit request by a user. Note that a user can create the effect of anticipatory fetching by simply issuing anticipatory reads for those applications (e.g., monthly payroll) whose future data requests are predictable.

The transfer of data into a level may necessitate the removal of some data already at that level in order to make room for the incoming data. This removal process is referred to as replacement and the replaced data is said to have overflowed.

The justification of a demand fetch policy is based on the fact that, for hierarchical storage systems, ". . . given any [series of requests] and replacement algorithm (not necessarily using demand paging) [a] replacement algorithm exists that uses demand paging and causes the same or a fewer number of pages to be loaded . . ." [20]. Intuitively, this means that demand paging leads to no more I/O requests within the hierarchy than any other possible algorithm. Since the number of I/O requests is a fundamental limiting factor for system throughput, using a policy which minimizes I/O requests is very attractive.

This policy leads to a retrieval scheme which operates roughly as follows:

1) a user issues a READ request

2) if the requested data is found in the cache level it is returned to the requesting user

3) if not found, the hierarchy is searched for the requested data. The requested data is transferred from the level in which it is found to the cache level, and from there to the requesting user.

This very general description leaves open a number of questions which will be addressed in the remainder of this chapter, including:

1) is only the requested data fetched or are entire blocks which may contain data which has not been explicitly requested retrieved?

2) if the data is blocked into pages, should the pages be the same size at all levels?

3) given that an entire page has been referenced, should a copy (or multiple copies) of the page be saved in the levels between the level at which the page was found and the cache level, in anticipation of future requests for data in that page?

4) how is the hierarchy searched - level by level or all levels in parallel?

5) finally, if this strategy results in an overflow, how should the page to be removed be selected, and what should be done with it?

Questions 1, 2, and 3 will be addressed in the next section, while questions 4 and 5 will be dealt with in Section 2.3.4.

2.3.3.2 Page Size Specification and Page Splitting

An examination of the random access times and transfer rates of the devices listed in Table 2.1 reveals that access times vary by over six orders of magnitude, while transfer rates vary by only two orders of magnitude. This fact, coupled with spatial locality considerations, can be used to show that system performance can be optimized (with respect to total expected data transfer delay) by

1) a choice of page sizes such that $N^1 < N^2 < \ldots < N^r$ (where N^i is the size of the unit of transfer from Level i+1 to Level i, and also the size of the page stored at Level i) coupled with

2) the use of a page splitting algorithm to determine the placement of data in the various levels in the hierarchy.

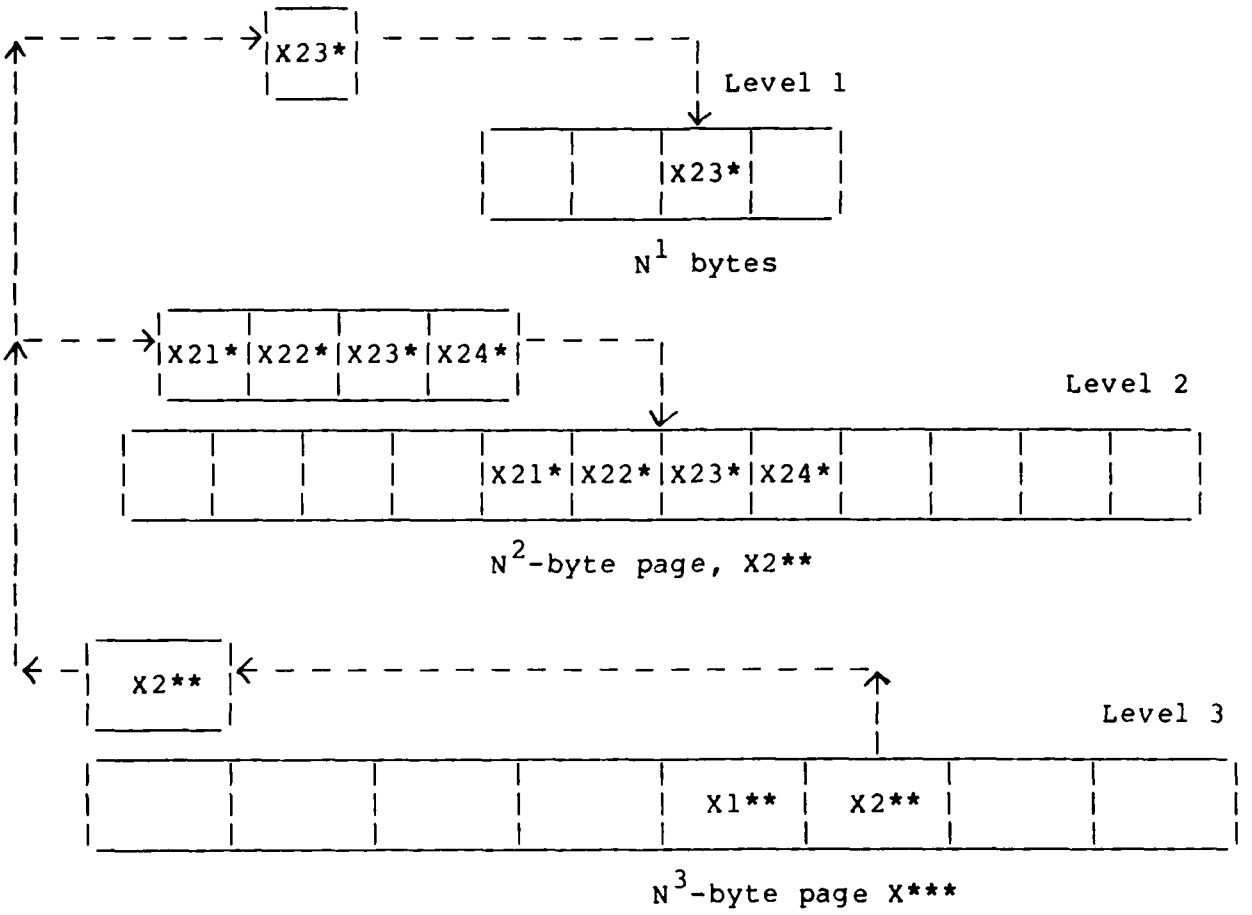
(For a detailed derivation of this result, see [17].) Intuitively, a factor to consider in the choice of N^i is that this choice represents a tradeoff between the sensitivity of the system to spatial and temporal locality, respectively. A smaller page size decreases the sensitivity to spatial locality (less spatially related data is retrieved by each fetch), but increases sensitivity to temporal locality by allowing a level to hold a larger and more diversified collection of pages. Of

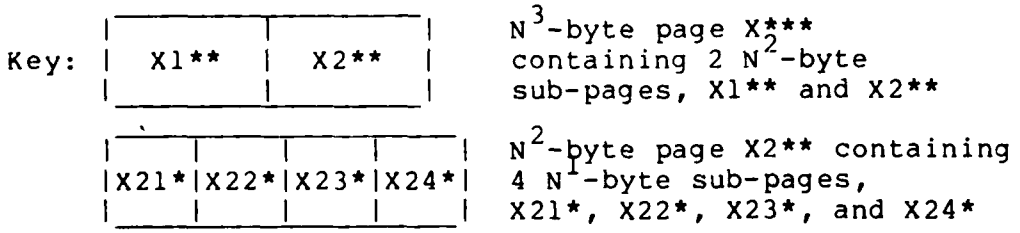
course, the optimal values of N^i will depend on the actual degree of locality and the speeds of the various storage devices in the hierarchy.

In order to retain as much flexibility as possible in the design of DSH-III, we allow the size of the unit of transfer between Level 1 and the user to differ from the page size, N^1, at Level 1. This idea is consistent with the variation of page sizes within the storage hierarchy itself. The size of the data blocks transferred, via the UBUS, between DSH-III and a user will be denoted N^0.

Page splitting operates as follows. Suppose a referenced data item is found in some page, of size N^3, at Level 3. The sub-page (of size N^2) containing the referenced data item is retrieved and transferred to Level 2 via the GBUS. Logically, the next step would be for Level 2 to transfer the sub-page of size N^1 containing the referenced data item to Level 1. In practice, this is not necessary, since Level 1 can obtain the appropriate sub-page from the GBUS during the initial transfer of data from Level 3 to Level 2. Therefore, the strategy actually used by DSH-III is for the level at which the data is found to "broadcast" the appropriate page over the GBUS, with each level extracting the appropriate sub-page from the broadcast data. This strategy is called READ-THROUGH since the requested data is read through from the level at which it is found directly to the highest storage level. This procedure is illustrated in Figure 2.3.

This figure gives an example of the notation we will use when it is desired to make explicit the relationships between a set of pages and their sub-pages at various levels. Virtual addresses of the smallest addressable units of data, i.e., pages of size N^0 bytes, will





Note: Pages not drawn to scale, i.e., $N^3 > N^2 > N^1$

Figure 2.3 - Illustration of Page Splitting as Page X23* is Broadcast From Level 3 to Levels 1 and 2

be denoted by sequences of letters and/or numbers. Thus, X, 12345, and AB3 might denote addresses. The addresses of pages at Level i are

denoted by a sequence of letters and/or numbers followed by i asterisks. Thus A** denotes the address of a page in Level 2. Page/sub-page relationships are expressed by using identifiers with all non-asterisk symbols equal. Thus A** contains (is the "parent" of) AB* (its "child"). Page AB*, in turn, is the parent of AB3. In situations where the relationships between pages are clear from the context, we will adopt the simpler notation of denoting pages by upper case Latin letters, e.g., X, Y.

In order to simplify implementation of the READ-THROUGH strategy, it is desirable that the page sizes be powers of 2. From now on we will assume that $N^i = 2^{k_i}$. This means that each page in Level i contains an integral number of sub-pages of size N^{i-1}. The total capacity of Level i, denoted C^i, can be computed as $C^i = m^i N^i$, where m^i is the number of pages of size N^i in Level i.

2.3.3.3 Page Splitting and Redundant Data

One result of using a page splitting strategy is that the system will contain redundant copies of data blocks. As a data item is read through into Level 1 it leaves a copy of itself, embedded in an appropriately sized page, in each level of the hierarchy.

There are two disadvantages associated with this redundancy. First, from a myopic point of view, redundancy is wasteful of storage space, but this ignores the performance gains resulting from a page splitting policy. One alternative to page splitting would be to transfer the requested data, X say, to Level 1 only (in a page of size N^1). Then, as subsequent references caused other pages to be brought into Level 1, X would eventually overflow and would be moved down to

Level 2 to make room for the incoming data. This type of overflow handling policy will be justified in the discussion of overflows in Section 2.3.4.3. Moving X from Level 1 to Level 2 would result in two I/O's and a bus transfer. In addition, since the pages in Level 1 are N^1 bytes and the pages in Level 2 are $N^2 > N^1$ bytes, the rest of the page of size N^2 containing X will have to be retrieved and moved into Level 2 in order to maintain the consistency of the paging scheme at Level 2. This latter retrieval will result in two more I/O's and another bus transfer. This process would then have to be repeated as the page removed from Level 2 to make room for X is moved to Level 3, and so on, all the way down the hierarchy. It is clear that this scheme involves considerable overhead, both in terms of extra I/O's and added bus traffic. It will be shown that the redundancy caused by page splitting allows the use of READ and WRITE algorithms which eliminate this overhead by discarding removed pages while still obtaining the performance benefits due to considerations of temporal locality.

The second disadvantage arises out of the potential for inconsistencies present in any system with redundant data. It will be shown that the algorithms used by DSH-III eliminate this possibility.

There are also some positive aspects to the data redundancy in DSH-III. Besides the performance advantages alluded to above, redundancy enhances the ability of DSH-III to recover from failures and continue operation by reconfiguring itself to bypass a failed component.

Finally, note that maintaining redundancy does not have much impact on the total effective storage capacity of the hierarchy since, in any reasonable design, one would expect $C^i << C^{i+1}$.

The advantages of using a page splitting policy appear to outweigh the disadvantages. Therefore, this policy has been incorporated into the data movement algorithms of DSH-III.

2.3.4 READ Strategies

Up to this point we have discussed data movement strategies and the tradeoffs among various alternative policies at a fairly general level. After settling on a hierarchical structure and specifying possible data movement paths between levels of the system, various data management policies and tradeoffs were discussed. The strategies selected on the basis of this discussion included demand fetch with replacement, different page sizes at various levels (selected on the basis of storage device speeds and locality considerations), page splitting, and the storage of redundant data. The following sections further refine the specification of the READ algorithms used by DSH-III. Issues addressed include specification of how the READ-THROUGH policy operates, how pages are selected for replacement, and how overflows are handled.

2.3.4.1 Data Location and READ-THROUGH

If a referenced data item, X say, can not be located in Level 1 (the cache level), the rest of the hierarchy must be searched for X. There are two ways that this search might be implemented, serial or parallel. A serial search could operate as follows. The READ request is passed, via the GBUS, from Level 1 to Level 2. A directory in Level 2 is searched for a page containing X. If found, X is retrieved from the storage system in Level 2, and the READ-THROUGH process is

initiated to transfer X to Level 1 and the Functional Hierarchy. If X is not found in Level 2, the READ request is passed down to Level 3, and the search is repeated at that level, and so on, until the request has percolated down to a level which contains X.

A parallel search operates as follows. The READ request is broadcast from Level 1 to all lower levels of the hierarchy, which then perform simultaneous, parallel directory searches for X. The highest level to locate X then initiates the READ-THROUGH for X, after informing all the other levels that X has been located.

The parallel search scheme has an obvious advantage in that it has potential for minimizing the expected delay between initiating the search and locating the data. On the other hand, there are some potential drawbacks to the parallel search strategy. While it is true that the expected time for a parallel search is less than that for a serial search, the parallel search is wasteful of system resources in that only the results of one search will be used. All the other searches represent wasted effort. This would not be a concern if it was expected that there would be excess processing power at each level. However, the fundamental rationale for the multi-processor architecture of DSH-III is that throughput can be increased by processing multiple transactions in parallel. Thus the parallel search might use resources that would otherwise be employed doing "useful" work as a part of the inherent parallelism of DSH-III. There is an implicit assumption here that the search uses scarce system resources. If, for example, the search was performed by specialized (and underutilized) hardware, there would be no disadvantage to parallel searches. Intuitively, the parallel search strategy represents an attempt to decrease response

time for individual transactions at the potential expense of decreasing overall system throughput. But note that the time for a serial search grows linearly with the number of levels that must be searched, while the data retrieval times grow by orders of magnitude as one moves down the hierarchy. This implies that the parallel search strategy is relatively less advantageous for retrieving data from lower levels of the hierarchy because the total search time for either strategy is relatively insignificant compared to the data retrieval time at lower levels. Therefore, for those retrievals where the parallel strategy has the greatest absolute advantage, the relative impact on response time is small. In addition, locality considerations lead one to expect that the majority of retrievals will be satisfied at high levels of the hierarchy. For retrievals from these levels, the serial search time is not much greater than the parallel search time, and thus does not have a great impact on response time.

Another drawback of the parallel search strategy is the extra algorithm complexity and control protocol overhead needed to coordinate the searches at different levels in order to determine the highest level at which the data was found.

The tradeoffs between serial and parallel searching will be the subject of further investigation, since the relative merits of the two schemes are dependent on the exact hardware configuration and load on the various components of the system. For the purposes of this paper, we will use the serial search strategy because it is simpler and because the parallel strategy does not seem to offer any significant performance benefits.

The precise mechanics of the READ-THROUGH process are fairly straightforward. Once the requested page has been located, it is retrieved from the LSS and broadcast to all higher levels of the hierarchy. Section 3 describes the architecture which allows the destination levels to extract appropriate sub-pages from the broadcast data, so that each receiving level obtains the correctly sized sub-page containing the originally referenced virtual address.

One potential way to decrease system response time would be to order broadcasts so that data destined for higher levels was broadcast first. In particular, the page destined for the cache level (and the Functional Hierarchy) could be broadcast first. In other words, the first data broadcast would be the sub-page of size N^1 containing the referenced virtual address. Next, the sub-page of size N^2 containing the referenced data would be broadcast, and so on until the entire data block had been broadcast.

2.3.4.2 LRU Replacement

The next issues to be addressed deal with the choice of a replacement policy and the design of overflow handling algorithms.

DSH-III will use a Least Recently Used (LRU) replacement policy. At any point in time, the pages at a level can be ordered by time of most recent reference. This ordering is called the LRU stack for the level, and each page at the level occupies a unique position in the stack (which will change as references are made to the level). Under the LRU policy, the page at the bottom of the stack (i.e., the page least recently referenced) is the one selected for removal. There are three reasons for the selection of LRU as the replacement algorithm for

DSH-III:

1) LRU has been shown empirically to compare favorably with the "optimal" removal algorithm, MIN [5]. (MIN itself can not be used as a replacement algorithm since it requires knowledge of the future.)

2) LRU is one example of a class of "stack algorithms" [20]. These algorithms can be shown to have "inclusion" properties which are used to simplify the data movement algorithms of DSH-III.

3) One consequence of the inclusion property of LRU is that the "hit ratio" for a level (the fraction of READ requests satisfiable at that level) is a monotonic function of level size. Thus LRU is not subject to certain anomalies [6] which can decrease performance as level size increases.

Now suppose that a reference to some data item, X, is satisfied at some level, j (the highest level containing X). This will cause X to be moved to the top of the LRU stack at Level j. Then the READ-THROUGH for X will result in a copy of X being stored in all levels, i, with $i < j$, and all these levels will have their LRU stacks changed to reflect the fact that the most recent reference at each of these levels was a reference to X. The algorithm described up to this point is referred to as LOCAL-LRU [15]. It is characterized by LRU stack updates being performed only at those levels which actively participate in a READ-THROUGH, in this case Level 1 to Level j.

An alternative policy is termed GLOBAL-LRU. Under this policy, the LRU stacks in Level 1 to Level j would be updated as for LOCAL-LRU. In addition, the LRU stacks in all levels below Level j which contained X would also be updated just as if a reference to X had been made at each of those levels.

Figures 2.4 and 2.5 illustrate these two alternative LRU policies. The justification for the choice of LRU policy used by DSH-III will be deferred until the issue of overflow handling has been discussed.

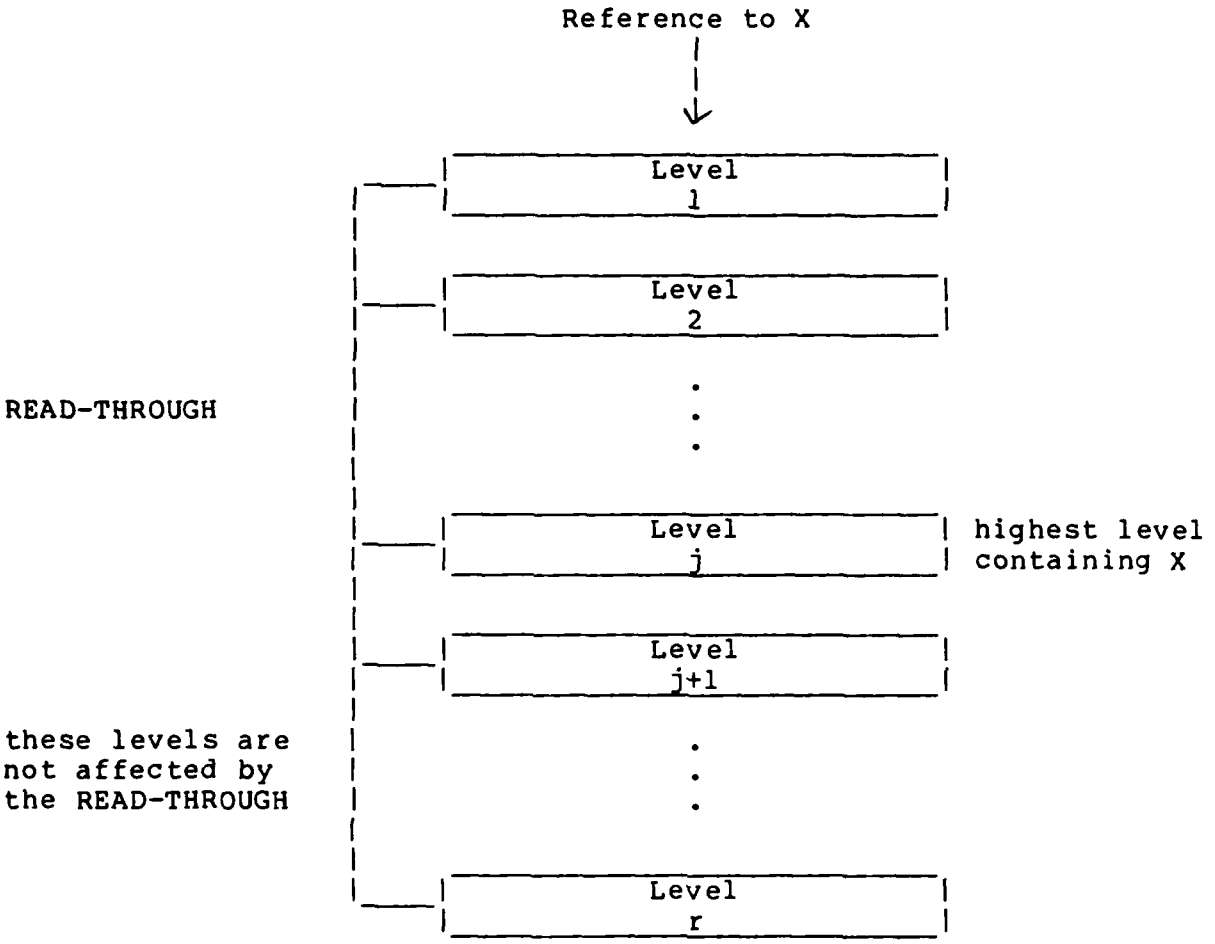


Figure 2.4 - Illustration of LOCAL-LRU Algorithm

2.3.4.3 Overflow Handling

When an overflow occurs there are three possible courses of action:

option 1: The overflow page could be simply discarded. (Since a copy of every page exists in the reservoir, this policy does not lead to the loss of any data.)

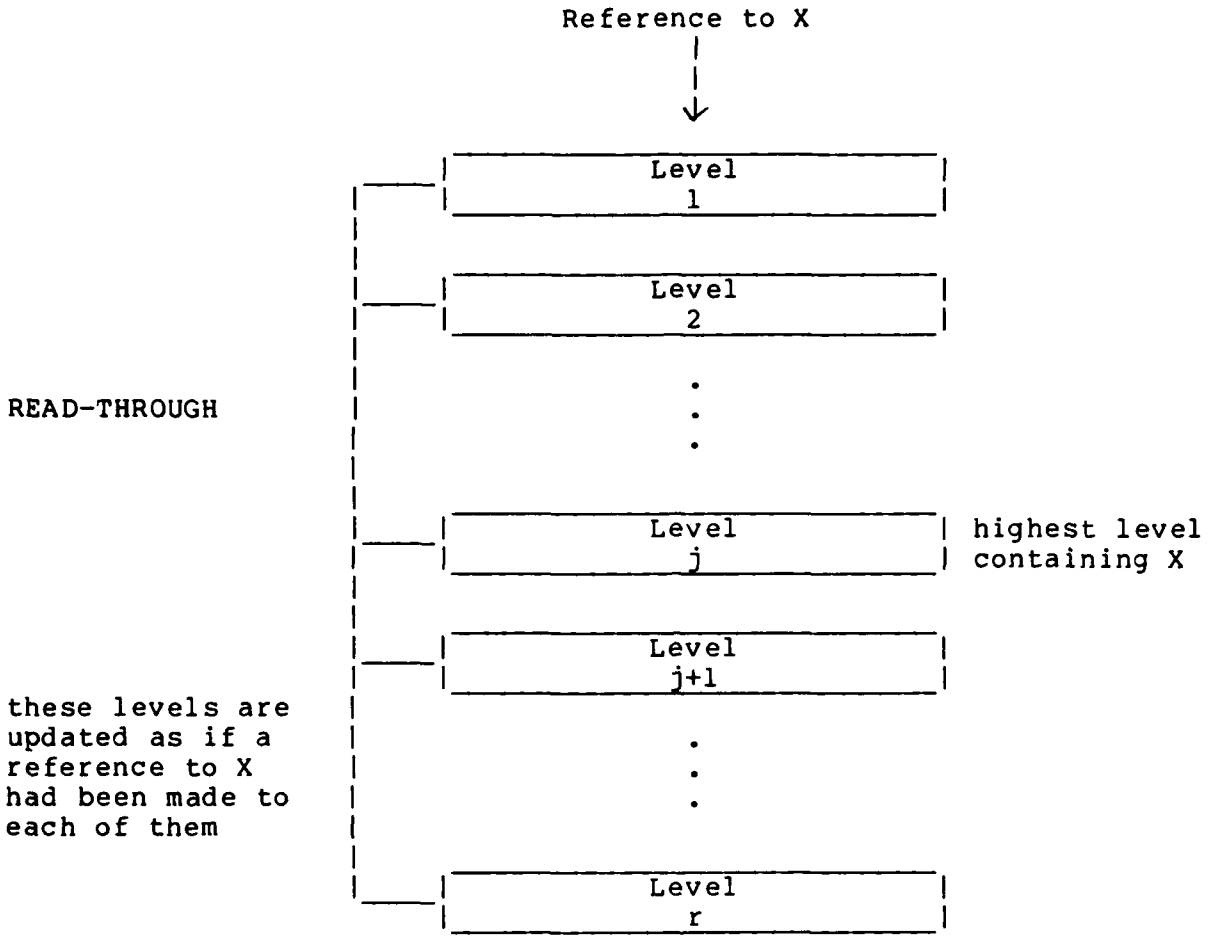


Figure 2.5 - Illustration of GLOBAL-LRU Algorithm

option 2: The overflow page could be moved to the next lower level of the hierarchy.

option 3: The overflow page could be moved to some other location(s) in the hierarchy.

Before presenting the pros and cons of these alternatives, let us recall the original rationale for adopting a hierarchical system design. The idea was to create a structure which would take advantage of locality by providing a range of storage devices and allowing the dynamic allocation of data to faster or slower devices based on

anticipated future usage patterns. Therefore, we adopt as a "preferred" overflow policy one which attempts to keep more recently referenced data in higher (and faster) storage levels. The obvious way to accomplish this is to maintain an LRU stack for the entire data base, and to allocate pages to higher or lower levels according to their position in this global LRU stack. Clearly, this is infeasible, since a global LRU stack runs contrary to the principle of distributed control and hierarchical decomposition. Instead, we can approximate the ideal strategy by maintaining separate LRU stacks within each level, and moving pages down the hierarchy one level at a time each time they overflow.

This implies that option 2 should dominate option 3, so option 3 is discarded as a potential overflow handling strategy. Thus the choice of overflow policy is reduced to selecting either option 1 or option 2 on the basis of degree of conformance with the "preferred" policy (subject, of course, to performance considerations).

In order to define option 2 completely, we must specify what effect an overflow from Level i has on the LRU stack at Level i+1. Lam [16] proposes two algorithms for handling this situation: Static Overflow Placement (SOP) and Dynamic Overflow Placement (DOP). Under an SOP policy, the overflow of a page, X, from Level i has no effect on Level i+1 unless there is no copy of X at Level i+1, in which case the overflow is treated as if a reference to X had been made at Level i+1. Under DOP, an overflow of a page, X, from Level i results in the LRU stack at Level i+1 being updated as if a reference to X had been made at Level i+1.

Two things should be noted regarding these policies. First, under either SOP or DOP, if X is not in Level i+1, the reference generated by the overflow results in a READ-THROUGH of X to Level i+1. Therefore, in this case SOP and DOP have identical effects but involve significant overhead. Second, if X is found in Level i+1, SOP implies that no action need be taken beyond verifying that X is indeed in Level i+1. Notice that in this case (option 2 with SOP and X found in Level i+1), option 1 and option 2 are equivalent, except for the verification that X is in Level i+1.

Tables 2.2a and 2.2b summarize the pros and cons of the three overflow handling options in the light of the foregoing discussion.

	Option 1	Option 2 (SOP)	Option 2 (DOP)
Conformance with Preferred Policy	low	high	high
Bus Load	low	high	high
Complexity	low	moderate	moderate

Table 2.2a - Overflow Policy Tradeoffs (No Redundancy)

	Option 1 and Option 2 (SOP)	Option 2 (DOP)
Conformance with Preferred Policy	high	high
Bus Load	none	low
Complexity	low	moderate

Table 2.2b - Overflow Policy Tradeoffs (Overflow Inclusion)

Table 2.2a assumes that there is no redundant data in the system, while Table 2.2b assumes that the parent page of any overflow page will always be in the next lower level. We term this property Overflow Inclusion. This distinction is important because, as can be seen from Tables 2.2a and 2.2b, Overflow Inclusion eliminates the need to perform any actual data transfers in order to support any of the three overflow handling options. With Overflow Inclusion, the only difference between the three options is the need, under option 2 with DOP, to update the LRU stack at Level i+1 in the event of an overflow from Level i. Notice that the assumption of Overflow Inclusion eliminates the need, under SOP, to determine whether or not an overflow page has a copy in the next lower level.

Tables 2.2a and 2.2b show that the "best" overflow handling scheme can be achieved in a system with Overflow Inclusion. Without Overflow Inclusion, the choice is between a policy (option 1) which is simple and involves little overhead but does not have the desirable features of the "preferred" policy and a policy (option 2) which conforms to the "preferred" policy but imposes heavy overhead on the system. The next section shows how Overflow Inclusion can be achieved at little cost in terms of overhead or complexity by simply placing loose bounds on m^i.

The original rationale for DOP (as opposed to SOP) was that DOP conformed more closely with the "preferred" overflow concept, in that an overflow from Level i would move from the last slot in the LRU stack at Level i to the first slot of the LRU stack at Level i+1. This agrees with the view of the stack at Level i+1 being a logical extension of the stack at Level i. A closer examination reveals that this advantage of DOP over SOP is largely illusory, if GLOBAL-LRU is

used. Assuming that there is redundancy in the system, the effect of DOP is to move X, the overflow from Level i, ahead (in terms of LRU stack position in Level i+1) of all the pages in Level i+1 which contain sub-pages in Level i. Thus the page, X, which is no longer in Level i, is, in some sense, being treated as if it were more recently referenced than pages which are still in Level i. Of course, when these latter pages eventually overflow from Level i, the DOP algorithm will restore them to their "rightful" place ahead of X in the LRU stack in Level i+1. However, Theorem 4 in the following section will show that SOP, in conjunction with GLOBAL-LRU, has approximately the same end result, with none of the complexity or overhead of DOP. Since we will be using an algorithm based on GLOBAL-LRU, SOP appears preferable to DOP as an overflow policy.

2.3.4.4 Multi-Level Inclusion (MLI) and Overflow Inclusion (MLOI)

If, at any instant of time, any page X in Level i is a subpage of some page in Level i+1, the storage hierarchy is said to have the Multi-Level Inclusion (MLI) property [15]. At first glance, it might seem that MLI is sufficient to guarantee the Overflow Inclusion mentioned in the preceding section, but this turns out to be false. Consider the three level hierarchy shown in Figure 2.6a. In this hierarchy, the same data, X, is in the next page to be selected for eviction from both Level 1 and Level 2. Figure 2.6b shows the effect of a READ-THROUGH from the reservoir in this situation. As a result of the READ-THROUGH, Level 1 and Level 2 are updated simultaneously, resulting in a simultaneous overflow of X from each level. Thus at the instant that X overflows from Level 1, there is no copy of X in Level

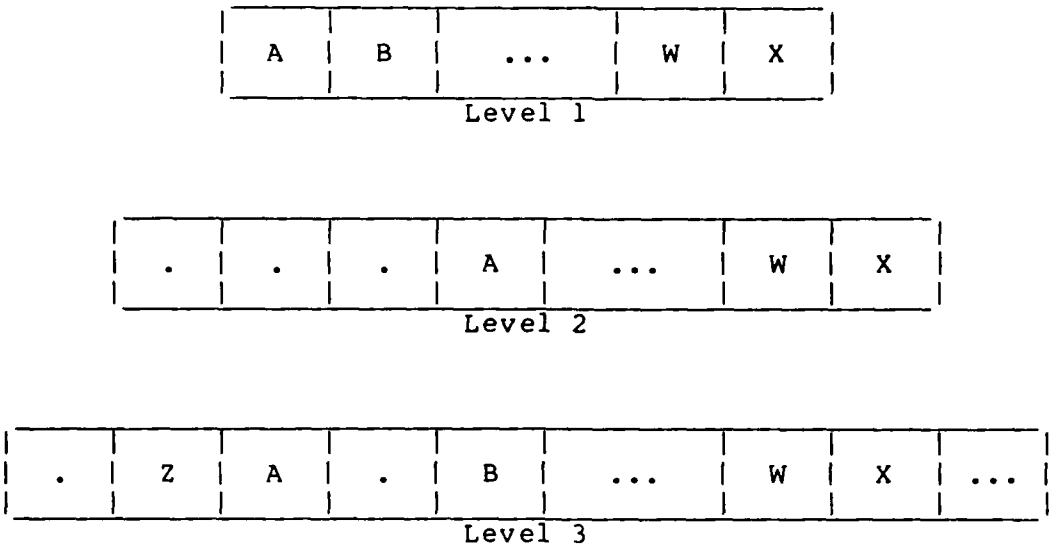


Figure 2.6a - Hierarchy Just Before READ-THROUGH for Z

2, even though the MLI property holds for this hierarchy. In this case, discarding the overflow page would leave no copy of X in Level 2, an undesirable outcome from the point of view of taking full advantage of temporal locality. An inclusion property slightly stronger than MLI is needed to prevent this situation from occurring. This property is Multi-Level Overflow Inclusion (MLOI). MLOI holds if any overflow page from Level i is a subpage of some page in Level i+1 and MLI holds as well. Thus, MLOI is sufficient to allow overflows to be discarded under any reasonable overflow handling policy. The following sections describe the conditions under which the MLI and MLOI properties can be guaranteed to hold, and briefly discuss the implications of these properties for performance and reliability of DSH-III.

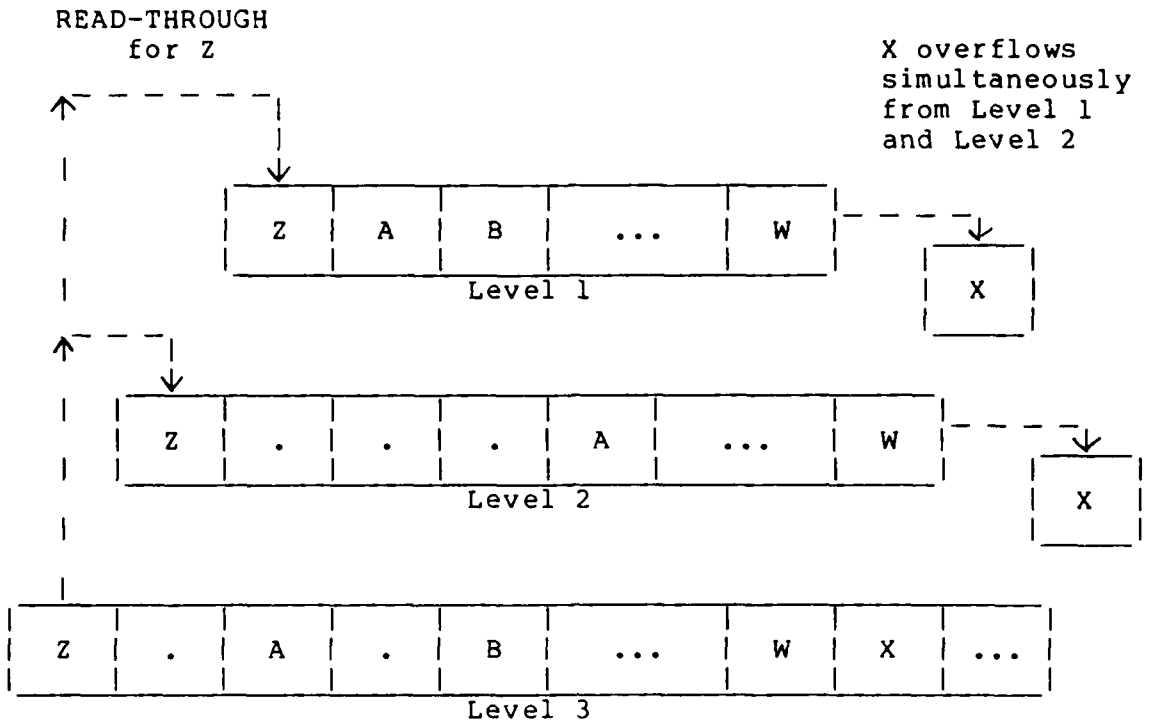


Figure 2.6b - Hierarchy Just After READ-THROUGH for Z

2.3.4.4.1 Theoretical Basis for MLI and MLOI

In order to define precisely the conditions under which MLI and MLOI hold it is necessary to completely specify the data movement algorithms used by DSH-III. The assumptions that have been made so far regarding what would constitute a "good" algorithm are:

- the use of READ-THROUGH with page splitting
- the use of an LRU replacement policy at each level
- the "preferred" overflow handling policy involves logically moving an overflow page to the next lower level. Recall that if MLOI holds, this policy imposes no overhead on the system, since no actual data movement is needed.

Given these basic assumptions, there are four possible READ strategies that can be derived by selecting either LOCAL- or GLOBAL-LRU in combination with either Static Overflow Placement (SOP) or Dynamic Overflow Placement (DOP). Table 2.3 shows the four possible strategies.

		Overflow Handling Policy	
		Static Overflow Placement (SOP)	Dynamic Overflow Placement (DOP)
LRU Policy	LOCAL-LRU	LOCAL-LRU-SOP	LOCAL-LRU-DOP
	GLOBAL-LRU	GLOBAL-LRU-SOP	GLOBAL-LRU-DOP

Table 2.3 - The Four Possible READ Algorithms

We now present a series of theorems which characterize the inclusion properties of hierarchical storage systems using these four algorithms [16].

Theorem 1: Using any of the four algorithms, if $m^1 \geq 2$ and $m^j \leq m^{j-1}$ for some j then there exists a reference string which leaves the system in a state where MLI does not hold.

This theorem implies that, if MLI is to be guaranteed, m^j must be strictly greater than m^{j-1} for all j. In other words, the number of pages in each level must be greater than the number of pages in the next higher level. This is not a restrictive condition since one of the precepts of the hierarchical design is to have the capacities of the levels increase from top to bottom of the hierarchy.

Theorem 1 gives necessary conditions for MLI (and therefore MLOI) to hold, but these conditions are not sufficient, as is shown by the next theorem.

Theorem 2: Using LOCAL-LRU-SOP or LOCAL-LRU-DOP, if $m^1 \geq 2$, there exists a reference string which leaves the system in a state where MLI does not hold.

Based on this theorem, we reject the two algorithms using LOCAL-LRU as potential candidates for DSH-III.

The next theorem gives conditions on m^i which guarantee that MLOI (and therefore MLI) hold for all possible reference strings for the two algorithms using GLOBAL-LRU.

Theorem 3: Using GLOBAL-LRU-SOP, if $m^1 \geq 2$, MLOI holds for any reference string if and only if $m^j > m^{j-1}$. Using GLOBAL-LRU-DOP, if $m^1 \geq 2$, MLOI holds for any reference string if and only if $m^j \geq 2m^{j-1}$.

This theorem gives fairly loose conditions (especially for GLOBAL-LRU-SOP) on the relative sizes of the levels of a hierarchy which are sufficient to guarantee that MLOI always holds.

Thus if a hierarchy is subject to the constraints of Theorem 3, we can implement a READ strategy based on either GLOBAL-LRU-SOP or GLOBAL-LRU-DOP, which constitutes a "good" algorithm based on the arguments presented so far in this paper.

2.3.4.4.2 Performance Implications of Maintaining MLI and MLOI

The effects on performance of a policy which attempts to maintain MLI and MLOI can be conveniently partitioned into benefits and drawbacks. The benefits have been discussed above, and are briefly

summarized here. They include

- support for a "preferred" overflow policy with no attendant data transfer overhead
- conformance with the principle of using varying page sizes in conjunction with page splitting in order to minimize expected retrieval times
- enhancement of system availability by allowing intentional (e.g., for preventive maintenance) or unintentional (e.g., in case of failure) removal of a level without changing the logical structure of the system or its data movement algorithms.

The drawbacks associated with maintaining MLI and MLOI arise from two sources

- extra complexity and processing overhead within each level
- extra interlevel communication overhead

Each of these sources will contribute to a degradation in performance in varying degrees, depending on the exact policy used to maintain MLI and MLOI.

As an example of one such policy, the reader is referred to Lam's approach [15], which presents a set of READ and WRITE algorithms which maintain MLI. This approach is based on the idea of associating with each page at each level a USC (upper storage copy) flag which indicates whether or not a sub-page of the page is resident in some higher storage level. Lam uses a modified LRU replacement policy which selects the least recently used page which does not have its USC flag set as the candidate for replacement.

In order to maintain the correct USC flag values, a level must notify the next lower level whenever it evicts the last sub-page of some page, X. This policy is complicated by the possibility that the notification of the eviction of the last sub-page of X could occur simultaneously with a READ-THROUGH for some other sub-page of X. This

situation is called "erroneous overflow". Delicate algorithms are needed to ensure that the notification is cancelled appropriately in this case (and in other pathological cases which can arise). In fact, Lam's original algorithms do not handle all possible cases correctly, and must be modified slightly in order to prevent erroneous USC flag settings.

Lam's algorithm has the advantage of being conceptually simple (i.e., it is intuitively obvious that this algorithm does, in fact, maintain MLOI) but it does entail some communication overhead (the USC notification messages) as well as some processing overhead (in order to determine when the last sub-page of some page is being evicted). In addition, as noted above, there are a number of subtle pathological cases such as "racing requests", "erroneous overflows", and "overflows to partially assembled blocks" that must be dealt with.

The algorithms proposed in this paper attempt to maintain MLOI by implementing GLOBAL-LRU-SOP subject to the conditions of Theorem 3, above. In essence, MLOI is obtained as a by product of the data movement algorithm, and therefore we do not need to consider any of the pathological cases that greatly increase the complexity of Lam's approach. The advantages of the approach presented herein include

- no overhead for overflow handling
- avoidance of much of the computational complexity implied by Lam's algorithms.

On the other hand, this approach has some disadvantages, including

- a need to perform strict LRU replacement
- a need to synchronize LRU updates between levels (in order to perform GLOBAL-LRU properly)

- restrictions on the relative sizes of the levels of the hierarchy, as specified by Theorem 3 (although, as noted before, these restrictions are not constraining on any reasonable design, that is, one for which the number of pages per level increases from Level i to i+1). Furthermore, this restriction is implicit in Lam's algorithms since obviously MLI can not be maintained if $m^j < m^{j-1}$ for any j.

Table 2.4 summarizes the comparison between Lam's READ algorithms and the ones proposed in this paper.

	Lam's Algorithms	Proposed Algorithms
Replacement Policy Restrictiveness	unrestrictive	fairly restrictive
Replacement Policy Complexity	fairly high	fairly low
Overflow Handling Overhead	moderate	none
Constraints on Hierarchy Structure	loose	loose

Table 2.4 - Comparison of Two Strategies for Maintaining MLOI

The only significant advantage of Lam's algorithms appears to be in the first category, replacement policy restrictiveness, while its only significant disadvantage is in the second category, replacement policy complexity. Thus a choice between these two policies will turn on which of the two categories has the greatest impact on performance. In point of fact, Lam's simulation studies [15] showed that performance is limited by bus bottlenecks, rather than processing bottlenecks, although these studies appear to have been based on optimistic estimates of 1985 micro-processor technology. Both Lam's algorithms and those proposed herein represent an attempt to lower bus contention

(by reducing the page transfers needed to support overflow handling) at the expense of increased processing overhead and complexity. In light of Lam's simulation results, this general approach appears to be a plausible method of reducing the bus bottlenecks in the system. With this in mind, the proposed algorithms, based on GLOBAL-LRU-SOP, would seem to have an overall advantage, due to their lower bus utilization and less complex replacement policy. A final choice between the two policies will depend on the results of emulation studies to be performed using a proposed Software Test Vehicle (STV) [27]. On balance, however, a set of algorithms based on GLOBAL-LRU-SOP would seem to have almost all of the properties desirable in a data movement strategy, while still being quite simple and efficient.

2.3.4.5 Implementation Issues for GLOBAL-LRU-SOP

This section discusses a number of issues relevant to the implementation of a READ algorithm based on GLOBAL-LRU-SOP. For the sake of brevity, from now on this algorithm will be denoted GLS. The issues to be discussed are

1) pre-eviction of pages,
2) LRU update epoch selection,
3) LRU update synchronization, and
4) duplicate READ request handling.

2.3.4.5.1 Pre-eviction of Pages

Pre-eviction of pages refers to the process of selecting pages for replacement before they are explicitly forced out of a level by a READ-THROUGH (compare with post-purge used by Multics [21]). Of course, MLOI will be destroyed if a page with a sub-page in the next higher level is evicted in this manner. A pre-eviction algorithm which does preserve MLOI can be deduced from the following theorem.

Theorem 4: Define $S^j(X)$ to be the LRU stack position of page X in the LRU stack at Level j. Thus $S^j(X) = 1$ for the most recently referenced page, and $S^j(X) = m^j$ for the least recently referenced page in Level j. Using GLOBAL-LRU-SOP, if $m^1 \geq 2$ and $m^j > m^{j-1}$, then for any page X in Level j and sub-page, Y, of X in Level j-1, $S^{j-1}(Y) \leq k \Rightarrow S^j(X) \leq k$.

Proof: First note that the statement of the theorem can be written as $S^j(X) \leq S^{j-1}(Y)$. We will show that the theorem is true immediately after any reference to Y, and that succeeding references do not change the inequality. Immediately after Y is referenced, we have $S^j(X) = S^{j-1}(Y) = 1$. A succeeding reference either touches X, or it does not. If it does not reference X, then as a result of the reference both $S^j(X)$ and $S^{j-1}(Y)$ increase by one. If it does reference X, but does not reference Y, then, as a result of the reference, $S^j(X) = 1$ and $S^{j-1}(Y)$ increases by one. In either case, the inequality holds. QED

What this theorem says is that a page is always closer to the top of the stack than any of its children in higher levels. Based on this theorem, a page, X, can be safely pre-evicted from Level j as long as

$S^j(X) \geq m^{j-1} + 1$. Intuitively, what is going on is that pre-eviction is reducing the effective size of Level j, and as long as the effective size of Level j is not reduced below the limit specified by Theorem 3, MLOI will be preserved. This ability to dynamically alter the effective size of a level is a reflection of the "stack inclusion" property of stack algorithms, such as LRU [20]. Note that the criteria for pre-eviction at Level j, as specified by Theorem 4, depend only on the stack at Level j and the number of pages in Level j-1. Thus the pre-eviction algorithm does not depend on any dynamically changing information which is not local to Level j.

In practice, it will turn out that some pages in a level may be locked against eviction. This would be the case if, for example, a page was in the process of being retrieved in preparation for a READ-THROUGH, but the LRU update for the page had not yet been performed. In this case, it is not possible to evict the locked page. In order to prevent this situation from degrading performance, the pre-eviction algorithm does allow some deviation from strict LRU eviction by skipping over locked pages. Once again, appeal to the stack inclusion property of LRU shows that this process does not violate Theorem 3, and thus preserves MLOI.

One problem with pre-eviction is that if all pages, X, with $m^{j-1} < S^j(X) \leq m^j$ are locked, then there are no candidates for pre-eviction at Level j. Thus, Level j could run out of available page frames, and all READ-THROUGH's from below Level j would be blocked.

To see that this blocking phenomenon can not cause a deadlock, note that a level can only block levels below it. Therefore, Level 1 can never be blocked in this fashion, and can always accept a

READ-THROUGH (after waiting for a previous READ being satisfied at Level 1 to complete, if necessary). Thus, Level 2 can not be involved in a deadlock because it can always eventually initiate a READ-THROUGH to Level 1, thus freeing a page frame in Level 2. Similarly, this implies that Level 3 can never be involved in a deadlock, and so on for the rest of the levels in the hierarchy.

One final problem with pre-eviction is that, by reducing the effective size of a level, it degrades system performance. Thus, the ideal pre-eviction algorithm should strike a balance between not pre-evicting enough pages (thus potentially blocking READ-THROUGHs temporarily), and pre-evicting too many pages (thus reducing performance by reducing available storage at some level).

2.3.4.5.2 LRU Update Epoch Selection

It is possible that some time might elapse between the initial READ request and the reflection of the reference in the LRU stacks at the various levels. The GLS algorithm attempts to perform the LRU update as closely as possible in time to the point at which the actual READ-THROUGH of data to Level 1 is performed. This policy has been adopted so that the LRU stacks reflect as accurately as possible the sequence in which references are satisfied, rather than the sequence in which they are initiated. This policy seems to adhere most closely to the spirit of LRU replacement, and minimizes anomalies wherein a page has become a candidate for eviction from the cache before it has actually been retrieved. This would clearly be an undesirable situation.

2.3.4.5.3 LRU Update Synchronization

GLOBAL-LRU requires "simultaneous" LRU stack updates at each of the levels. In this context, simultaneous means that LRU updates should be seen in the same order by each of the levels in the hierarchy. These updates will be accomplished by broadcasting an LRU update message to all levels just prior to initiation of a READ-THROUGH. The READ-THROUGH itself can not be used as a synchronization signal because that would require that a READ-THROUGH be sent to all levels, not just higher levels.

2.3.4.5.4 Duplicate READ Request Handling

The final GLS implementation issue is duplicate READ request handling. Suppose two READ requests, for pages X1* and X2*, are received simultaneously, and both X1* and X2* are sub-pages of the same page, X**, of size N^2. Also suppose that X** is not in Level 2 (implying that neither X1* nor X2* are in Level 1, by MLI). Without loss of generality, assume that the request for X1* reaches Level 2 before the request for X2*. Then, when the request for X2* reaches Level 2, that level will already be expecting X** to be read through to Level 2, in order to satisfy the reference to X1*. We say that X** is "pending" at Level 2. Instead of forwarding the request for X2* to Level 3, Level 2 can hold the request until X** is transferred into Level 2, and then continue processing the request for X2* as if X** had been in Level 2 all along. This policy has been adopted in the expectation that the processing overhead involved in keeping track of pending requests will degrade performance less than processing duplicate requests (such as the one for X**) independently, and thus

incurring duplication of effort in the retrieval of X**. This policy also avoids the complication of having X** read through to Level 2 (as a result of the request for X2*) and finding a copy of X** already in Level 2 (as a result of the request for X1*).

2.3.5 WRITE Strategies

We now turn to a discussion of the WRITE strategies employed by DSH-III. It turns out that many of the problems encountered in designing suitable WRITE algorithms have already been addressed in the development of the READ algorithms for DSH-III. Indeed, many of the design issues mentioned previously were included in anticipation of their relevance to specification of WRITE strategies for DSH-III. Therefore, the brevity of this section is not an indication that WRITE is simpler than READ, but rather reflects the fact that READ and WRITE strategies have many common issues and problems which have already been dealt with. With this in mind, this section will concentrate on those issues particularly relevant to developing a WRITE strategy for a hierarchical storage system. These issues include reliability, update consistency, and buffer management. As always, the emphasis will be on maximizing performance, subject to constraints imposed by considerations of reliability, maintainability, and cost.

The WRITE process is initiated by a user issuing the command

WRITE(request_id,virtual_address,data).

This command is passed, via the UBUS, to Level 1 of DSH-III. At this point, Level 1 takes a number of actions in order to process the WRITE. These actions include

- making duplicate copies of the updated data in local memory at Level 1 in order to increase availability, i.e., in order to decrease the chance of losing an update if there is a (non-fatal) failure within Level 1,

- sending a "WRITE-complete" acknowledgement back to the user,

- ensuring that the N^1-byte page containing the virtual address to be updated is present in Level 1,

- possibly combining the update with other updates for the same page in Level 1, and

- initiating the process of transferring the update from Level 1 to the reservoir.

The remainder of this section will enlarge upon these points and justify the data management policies implied by them.

2.3.5.1 Initial Level 1 WRITE Processing

In order to guard against the possibility of lost updates, Level 1 will make a duplicate copy of every update. The copies of each update will be kept in independent memory modules at Level 1, thus providing protection against lost updates in the case of a memory failure at Level 1. Only <u>after</u> the update has been duplicated will a WRITE-complete acknowledgement be returned to the user. Thus, from a user's point of view, a WRITE will complete almost as fast as a READ which is satisfied at Level 1. The process of making a permanent copy of the update in the reservoir can now proceed asynchronously with the handling of subsequent user requests.

The first step in this process is to ensure that the N^1-byte page containing the virtual address to be updated is present in Level 1. This is done by issuing a READ for that page, if it is not in Level 1. In most cases, however, the user will have read the page, preparatory to modifying it, and the page will already be in Level 1.

The second step in this process is to transfer the updated N^1-byte page from Level 1 to the reservoir.

2.3.5.2 Alternative Store Policies

We refer to the process of transferring an updated page from Level 1 to the reservoir as a Store process. There are four plausible Store policies which will be presented here. They are

- Store Through,
- Store Replacement,
- Store Behind, and
- Staged Store Through.

The function of any of these four policies is to transfer an updated N^1 byte page from Level 1 to Level r (or, more precisely, to an input buffer in Level r).

Before discussing the pros and cons of these policies, we introduce the concept of "coalescing" of updates. Suppose X* has been updated by the request

```
WRITE(request_id1,X1,new_value_of_X1)
```

Further suppose that the request

```
WRITE(request_id2,X2,new_value_for_X2)
```

arrives after the previous update to X* but before the Store process for X* has been initiated. Now, rather than initiating two Store processes for X*, the two updates, to sub-pages X1 and X2 of X*, can be coalesced and only one (twice-updated) copy of X* need be transferred to Level r. The coalescing of updates reduces the number of Store actions needed in this case by a factor of two.

Now consider a series of WRITE requests. In the short term, these requests can complete at a rate which is dependent mainly on the speed of Level 1. In this case, one can view Level 1 as a buffer between the user and the reservoir (which is the final repository for all updates). In the long run, however, the average throughput for WRITEs is limited by the speed of the I/O devices in the reservoir. The effects of this limitation can be mitigated in either of two ways:

1) reducing the number of updates reaching the reservoir, and

2) increasing (in some unspecified way) the effective size of the buffer between the user and the reservoir.

Point 2 has the effect of making the system less sensitive to transient peaks in the arrival rate of WRITEs, but does not address the fundamental limitation on the average arrival rate of WRITEs imposed by the long term rate at which the reservoir can absorb the updates. Point 1 does address this issue. For example, suppose k WRITEs per second is the highest rate which can be supported without coalescing. Then a coalescing algorithm which, on average, combined two WRITEs, as in the above example, would allow the system to support, in the long run, a gross WRITE arrival rate of 2k per second. The arrival rate of updates to the reservoir would be half this rate, or k per second, which, by assumption, is within the processing capacity of Level r. In general, if the system can coalesce an average of c WRITEs into each Store action, it will be able to support a gross arrival rate of WRITEs up to c times higher than the rate which could be supported with no coalescing. Since the ratio of READs to WRITEs is fixed for any application, this implies that a coalescing strategy can have a significant impact on overall system throughput. Of course, the overall improvement in throughput would be less than the factor of c

since coalescing does not effect the rate at which READs can be processed.

A final point ot be made regarding coalescing is that the degree of coalescing attainable (i.e., the value of c) is dependent on two factors:

1) the amount of time that elapses between the receipt of an update and the initiation of the Store for that update, and

2) the degree of WRITE locality exhibited by the system.

These two factors will play an important role in the selection of a suitable Store policy for DSH-III.

We now turn to a discussion of the advantages and disadvantages of the four Store policies.

2.3.5.2.1 Store Through

Under a Store Through policy, as each update is received, Level 1 broadcasts it to all the lower levels. Under this policy, there is no opportunity for coalescing at Level 1 and there is no flexibility in the choice of epoch at which the Store is performed. On the other hand, Store Through is inherently reliable, since the update is reflected to all levels immediately. Under this policy, we could dispense with duplicating the update at Level 1, as long as the WRITE complete acknowledgement was delayed until after the update had been broadcast.

Finally, note that each update reaches the reservoir as part of an N^1-byte block.

2.3.5.2.2 Store Replacement

Under this policy, an updated page is held in Level 1 until it is selected for replacement by the LRU replacement algorithm. It is then moved to Level 2, where it is held until selected for replacement, at which time it is moved to Level 3, and so on. As with Store Through, this policy completely restricts the choice of epoch at which the Store is performed, and has the added drawback of imposing a delay on each eviction operation. On the other hand, Store Replacement does provide a maximal "window" during which coalescing can take place, and, in fact, allows coalescing at each level as the update is moved down the hierarchy.

2.3.5.2.3 Store Behind

A Store Behind policy attempts to alleviate the major drawback to Store Replacement, while retaining most of the advantages. Store Behind is identical to Store Replacement except that the update is moved down from level to level whenever it is convenient (e.g., during idle bus cycles) rather than when the page is evicted.

An availability enhancing modification to Store Behind, called Two-Level Store Behind has been proposed [15]. Under this policy, each update is maintained in two adjacent levels, j and j+1 say, and not removed from Level j until the update has been propagated from Level j+1 to Level j+2. Thus, two copies of the update will exist at any time, providing protection against the failure of any single level. This sort of modification is clearly applicable to Store Replacement also.

Finally, note that both Store Replacement and Store Behind involve the transfer of increasingly larger blocks as the update moves down the hierarchy. Under either policy, each update reaches the reservoir as part of an N^{r-1}-byte block. These large update blocks could potentially lead to bus contention or buffer management problems at Level r.

2.3.5.2.4 Staged Store Through

Both Store Replacement and Store Behind take advantage of coalescing at every level, but could lead to excessive bus loads and/or Level r buffer space requirements due to the large data blocks being moved under either policy. Additionally, the size of the data blocks associated with each update reaching the reservoir could lead to excessive I/O loads on the storage devices at Level r. Store Through, on the other hand, involves the transfer of relatively small data blocks (assuming that $N^1 << N^{r-1}$), but does no coalescing, thus increasing the potential number of updates reaching the reservoir by an order of magnitude or more, depending on the degree of WRITE locality exhibited by the system. Staged Store Through represents a compromise attempt to combine the best features of Store Through and Store Behind. Under Staged Store Through, updated pages are held in Level 1 until it is convenient for them to be broadcast to all lower levels. Therefore, Staged Store Through can take advantage of coalescing at Level 1, while restricting the size of the update blocks reaching the reservoir to N^1 bytes.

It is also possible, under Staged Store Through, to perform some coalescing in Level r for updates which are in the Level r buffer awaiting transfer to permanent storage.

2.3.5.3 Evaluation of Alternative Store Policies

Table 2.5 summarizes the tradeoffs among the four Store policies discussed above.

	Store Through	Store Replacement	Store Behind	Staged Store Through
degree of coalescing	none	very high	high	fairly high
size of data block reaching level r	N^1	N^{r-1}	N^{r-1}	N^1
flexibilty of update epoch	none	none	high	high
algorithmic complexity	very low	moderate	moderate	low

Table 2.5 - Comparison of Four Store Policies

Based on these observations, the choice of Store policy would appear to be between Store Behind and Staged Store Through. The relative merits of these policies will depend on the degree of coalescing achievable in Levels 2 to r-1, the relative magnitudes of N^1 and N^{r-1}, and the relative speeds of the various levels. The STV will provide a framework for deciding between these two policies. The algorithms presented in this paper are based on Staged Store Through because it involves somewhat less algorithmic complexity.

We close this discussion by pointing out how each of Store Behind and Staged Store Through attempt to take advantage of the two strategies for increasing WRITE performance mentioned above, namely, reducing the effective number of updates and increasing the effective buffer size. Table 2.6 summarizes these concepts.

	Store Behind	Staged Store Through
reducing the number of updates reaching the reservoir	coalesces at each level	coalesces at Level 1 only
increasing effective size of the buffer between a user and the reservoir	uses entire hierarchy as a buffer	transfers updates in N^1-byte pages, as opposed to N^{r-1}-byte pages

Table 2.6 - Comparison of Performance Enhancing Strategies

2.3.6 Automatic Data Duplication

The final point to be discussed in this section deals with the issue of reliability. In the design of DSH-III, the reservoir represents a key component in terms of reliability. Reliance on the reservoir as the storage component of last resort can be viewed as both a liability and and asset in terms of overall system reliability. On the one hand, if a storage device in the reservoir fails, there is a potentially unrecoverable loss of database integrity. On the other hand, if reservoir failures can be avoided, then the databse will be preserved no matter what failures occur in other components of the system. Of course, in this case the system may be inoperative until the cause of the failure can be found and corrected, but once this has

been accomplished normal operation can resume with no loss of data. In light of this reasoning, it makes sense to support the designation of "critical" data for automatic data duplication.

Critical, in this sense, can refer to two types of data. The first type is that data which DSH-III itself decides is critical for its internal operation. This type of data might include page tables and other important system control information internal to DSH-III. The second type of critical data is that data which a user decides is critical. It is solely up to the user to decide which of his data is critical in this sense. DSH-III supports a primitive, SET_SECURE, which a user can use to designate ranges of virtual addresses for automatic data duplication.

For data which has been designated "critical", either by a user or by DSH-III itself, the system will perform automatic data duplication. DSH-III will maintain duplicate copies of any data so designated, and will automatically replicate the data if one of the copies is damaged by a failure.

This strategy should provide almost complete protection from loss of data or database integrity due to isolated component failures.

3 FUNCTIONAL DESIGN OF DSH-III

This section describes a proposed hardware architecture for DSH-III. Section 3.1 contains a brief summary of the rationale for the overall hardware design philosophy adopted for DSH-III. In Section 3.2, the functioning of the various classes of components comprising the system is described. Section 3.3 presents some preliminary reliability and availability estimates for the proposed hardware design, and shows how high availability can be attained by combining minimal hardware redundancy with an appropriate software strategy.

3.1 Overview of DSH-III Architecture

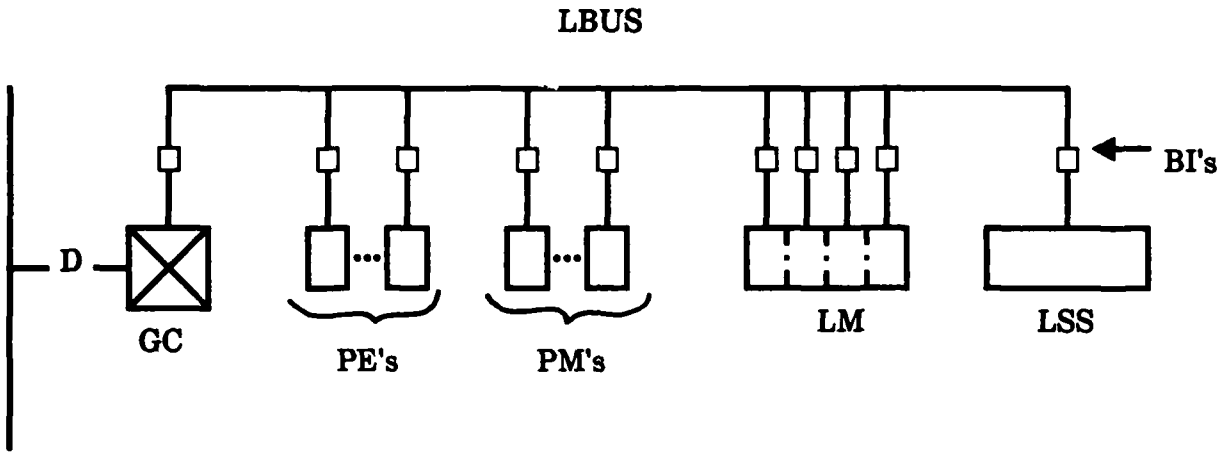
3.1.1 Overview of DSH-III Hardware

Figure 3.1 depicts the hardware design proposed for a typical level of DSH-III. Each level consists of a number of components of various types. In particular, each level will contain:

Processing Elements (PE's) - these components are general purpose micro-processors which will execute the system control and data management algorithms at each level.

Local Memory (LM) - this component consists of some number of high-speed semi-conductor memory modules, used for program and temporary storage at each level.

Processing Modules (PM's) - these modules will perform specialized hardware functions such as associative searches or hashed index calculations. (Initial versions of DSH-III will contain no PM's. They will be added if and when performance considerations justify their incorporation into



Hardware Architecture

Figure 3.1 Typical Level of DSH-III

the system design.)

Local Storage System (LSS) - this system will be used by each level to store portions of the virtual address space managed by DSH-III. In some cases, for example, Level 1 where data retrieval speed is a primary concern, the LSS may consist of the same storage technology as the LM. In fact, the LSS and the LM may actually share the same physical memory modules. In the interests of clarity of exposition, we have opted to retain the conceptual distinction between LSS and LM at each level, and will treat these components as if they were separate pieces of hardware. Figures 3.2a and 3.2b depict two possible LSS configurations, suitable for the "cache" and "reservoir" levels, respectively.

Local Bus (LBUS) - this is a high speed bus which will carry all intra-level communication traffic, including instruction fetches (from LM to PE) and data transfers (PE to LM, LSS to LM, and vice versa).

Gateway Controller (GC) - this component serves as the interface between the LBUS and the Global Bus (GBUS). As shown in Figure 3.3, the GBUS is a common data path which inter-connects all levels of DSH-III.

Bus Interfaces (BI's) - each one of the above components is connected to a bus by a Bus Interface. These interfaces are responsible for supporting the split transaction Pended Bus Protocol [28]. In effect, each BI creates the illusion of an infinite buffer between the device and the bus, with the result that the device never has to wait due to a bus busy

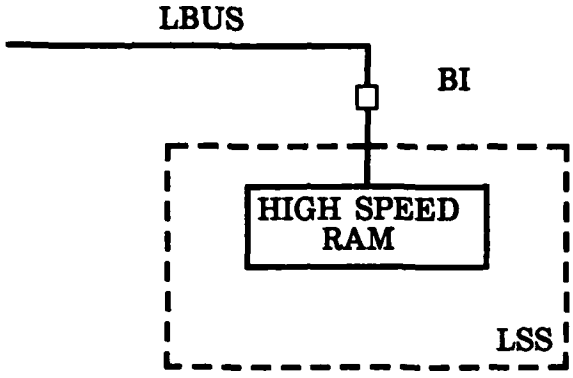


Figure 3.2a LSS at Level 1

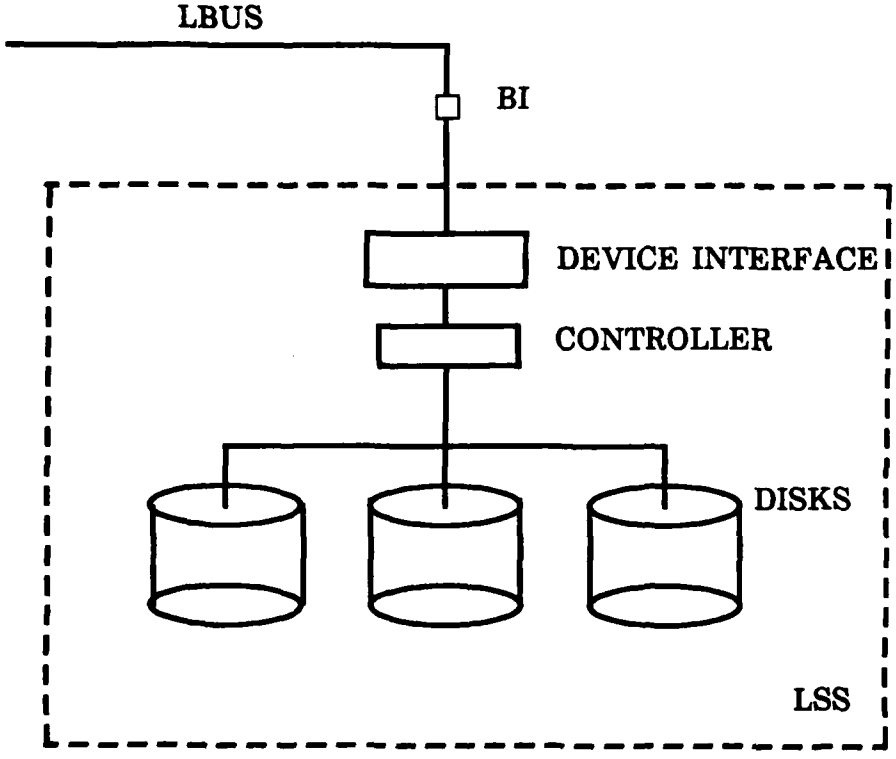


Figure 3.2b LSS at Level r

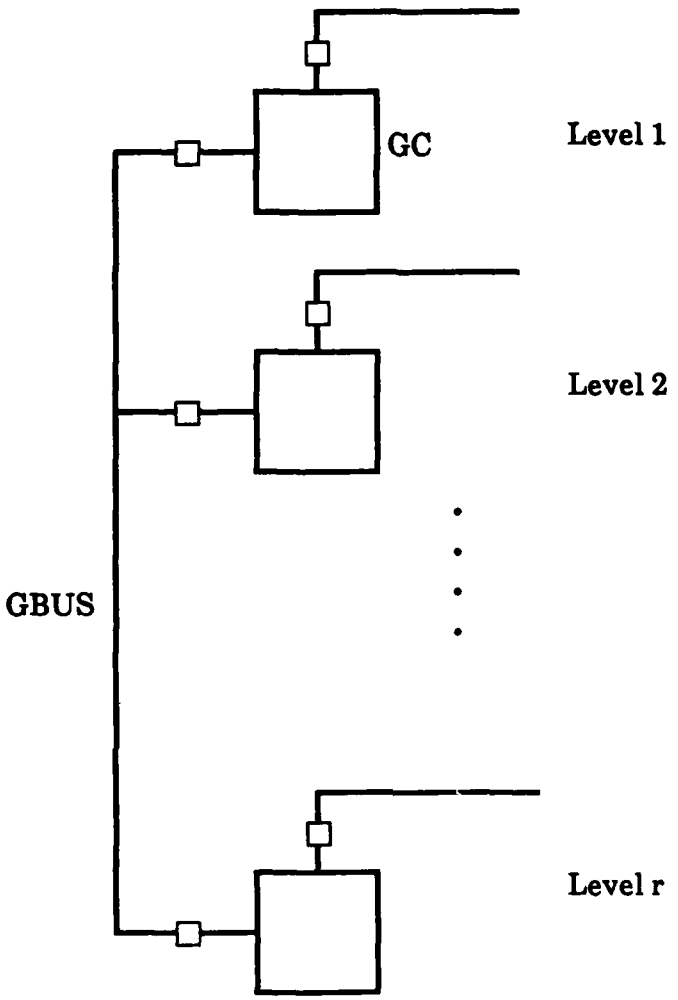


Figure 3.3 Interconnection of Levels by GBUS

condition.

In addition to all the components listed above, level 1 contains an additional GC which provides an interface between the level 1 LBUS and the User Bus (UBUS) which carries all traffic between DSH-III and its user(s). Figure 3.4 depicts a complete DSH-III system. Of course, in any actual implementation of DSH-III, key system components will be replicated in order to increase system availability. In particular, the LBUS's at level 1 and the reservoir, as well as the GBUS, might each be implemented as two, or more, physical buses since a failure of any one of these components would render DSH-III inoperative.

There are two fundamental reasons why this design was selected. First, the physical structure of the system corresponds closely to the structure chosen in Section 2 as being most suitable from purely logical considerations. The match between the logical and physical designs eliminates the need for a complex mapping of logical functions onto physical components and furthers the goal of being able to substitute alternate hardware components with only localized impact on the logical structure of the system.

The second reason for the selection of this design is that it represents the most cost effective way of providing the high bandwidth communication paths between various system components which are necessitated by the data management algorithms presented in Section 2. This design represents a cost/performance compromise between a design which provides a dedicated bus for every communication path and a single bus structure such as the one depicted in Figure 3.5. While the structure of Figure 3.5 is the least costly, its performance is severely degraded due to bus contention as more and more components are

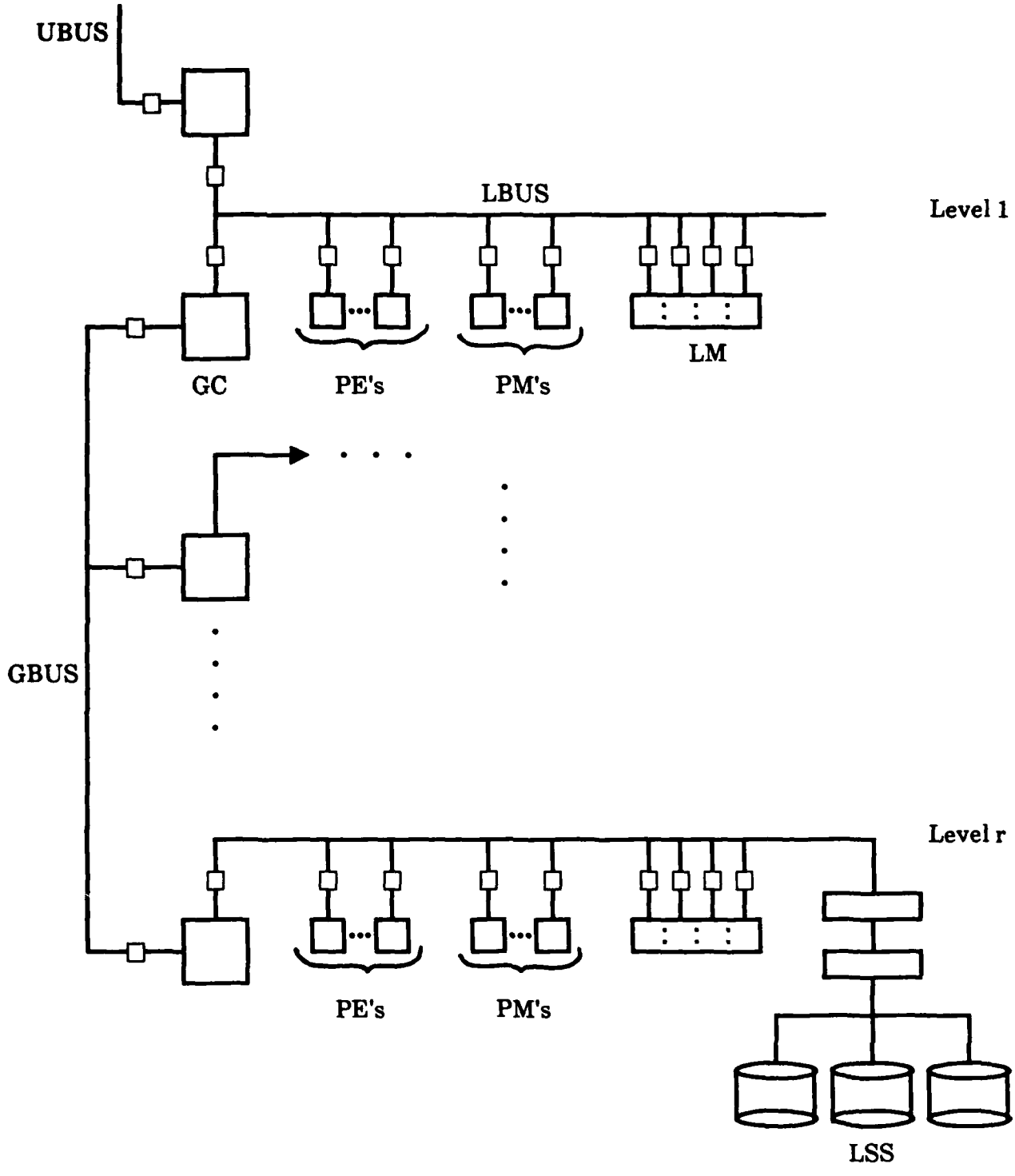


Figure 3.4 DSS-III Hardware Structure

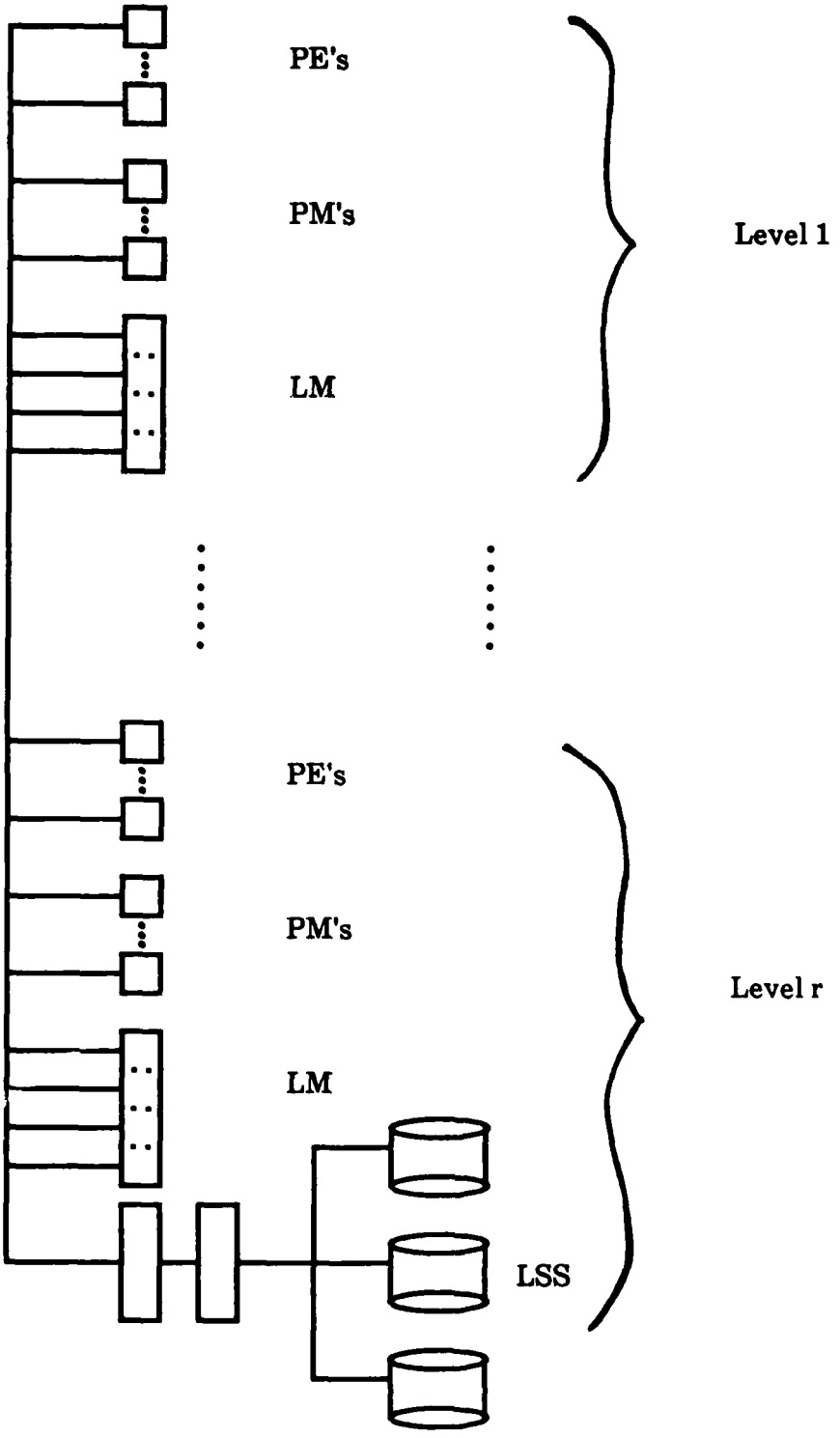


Figure 3.5 DSH-III as a Uni-Bus Structure

added to the system. Even if a Pended Bus Protocol is used, bandwidth limitations make a single bus structure unsuitable for DSH-III. The structure actually selected represents a compromise which allocates a dedicated bus (the LBUS) to the intra-level traffic at each level, and uses a separate bus (the GBUS) for all inter-level traffic. This structure, coupled with the use of the Pended Bus Protocol, represents a fairly parsimonious configuration which should provide adequate support for DSH-III.

A further advantage of this multi-level structure is that the processing power of each level can be adjusted by simply adding or removing PE's at that level. The next section describes a Local Operating System (LOS), responsible for processor management and resource allocation within each level, which allows this sort of hardware reconfiguration to proceed dynamically and independently of the algorithms currently executing at that level. This facility has a significant impact on system availability by allowing the incorporation of recovery algorithms which will make single-processor failures more or less transparent to the rest of the system.

3.1.2 Overview of DSH-III Local Operating System (LOS) Software

Each level of DSH-III operates completely independently of the other levels, with the only inter-level interaction being provided via the message passing facility described later in this section. Within each level, all PE's are treated as equivalent units with respect to both hardware and software. The PE's can be viewed as a set of identical resources which are used to advance the "algorithm" represented by the software and control information residing in the LM.

This software consists of both LOS and application software.

All software is written as if it were intended to run on a single processor, except for the inclusion of interlocks to protect writable shared data. From the point of view of an individual PE, its processing cycle consists of

1) executing the LOS dispatcher/scheduler which
   selects an eligible application task to be executed
   removes the selected task from the eligible list;
2) executing the selected task until
   the task completes; or
   the task becomes blocked waiting for some event, such as a message arrival, or waiting for a non-sharable resource to become available;
3) go to 1.

Assuming that appropriate interlocks are used to protect shared data, the above three steps can be executed by any number of processors simultaneously and independently.

This design should lead to a system which is more flexible and reliable than a system based on specialized processors. In effect, replication of hardware in order to increase processing power has the desirable side effect of increasing reliability at the same time. For an example of a working system based, in part, on these concepts, the reader is referred to [22].

3.2 Functional Characteristics of DSH-III Hardware Components

The previous section briefly introduced the various hardware components which are combined to produce a DSH-III system. In this section we will describe some of the key features of these components.

3.2.1 Processing Elements (PE's)

The PE's at each level will be identical general-purpose micro-processors such as Intel's iAPX 432 [10]. As noted above, from both a hardware and software viewpoint, the PE's within each level will be indistinguishable. The LM at each level will serve as program and data storage for each PE at that level. Each PE will independently execute a shared copy of the Local Operating System code which resides in the LM. The LOS will allow each PE to independently schedule its own activities. Task scheduling and processor allocation will be coordinated via shared databases which also reside in LM.

Conceptually, one can think of the address space of each PE as being composed of three separate segments, as shown in Figure 3.6. (Strictly speaking, one should refer to the address space of the level rather than the address space of a PE, since all PE's, being indistinguishable, share a single address space.)

Segment A consists of read-only program storage. This segment will contain all operating system and application program code for a level, and will reside in a write protected area of LM. One design alternative would be to place some or all of the read-only code in separate per-PE ROM's in order to reduce LBUS contention. This alternative will be further explored if experience shows that the LBUS is a significant bottleneck.

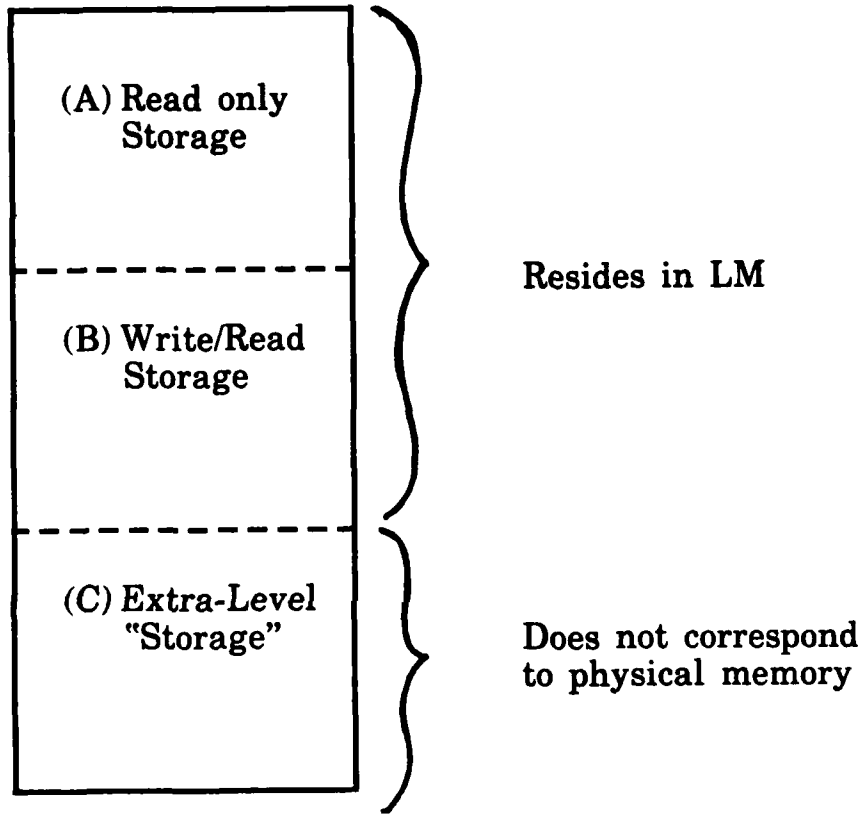


Figure 3.6 Address Space at a Level

Segment B differs from segment A only in that it is not write protected. This area will be used for program temporary storage, system control tables, etc., as well as for data storage for those levels that do not have a physically separate LSS. The coordination of access to this area is part of the memory management function of the LOS.

Segment C is a portion of the address space of each PE which is used for transferring data to other levels of the hierarchy. Segment C does not correspond to any physical storage device; rather, data which is stored into addresses which map into segment C is transferred directly to the GC which, in turn, moves the data onto the GBUS, where it is accessible to the GC's at the destination level(s). Figure 3.7 shows an addressing scheme which would reflect this architecture in a 32 bit micro-processor.

In this scheme, if the high-order bit of a destination address is set, the next n bits are interpreted as a bit map which indicates which of the n levels in the system should receive the data stored into the address. For example, suppose XB refers to some address in segment B, and that XC refers to address 10011100.... Then the instruction "MOVE XB,XC,32" might mean "transfer the 32 bytes, starting at address XB in LM, to levels 3, 4, and 5."

From the viewpoint of a PE sending a message, the data to be transferred is being stored directly into LM at the receiving levels, as if XC specified a physical address in the receiving level's LM. However, it is the responsibility of the receiving level to determine exactly where in the receiving level's LM the incoming message should be stored. The mechanism whereby the GC's operationalize this scheme

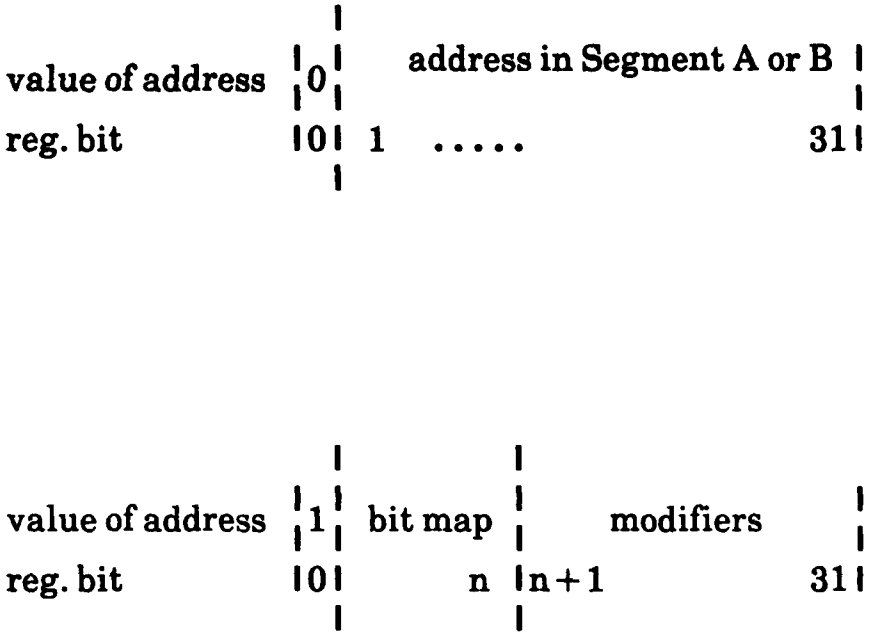


Figure 3.7--Addressing Architecture for DSH-III

is described in the next section.

3.2.2 Gateway Controller (GC)

The GC at each level serves as the interface between the LBUS at that level and the GBUS (and at level 1, between the LBUS and the UBUS, as well). For outbound messages, the GC merely moves the data bytes from the LBUS to the GBUS, and sets address lines on the bus as directed by the segment C address of the store instruction which initiated the message. It is the responsibility of the GC at the destination level(s) to extract the appropriate data from the GBUS (based on address lines set by the sending GC) and place it in the LM where it can be referenced by the PE's at the receiving level. The mechanism by which the GC determines where in LM to place inbound messages will now be described.

As currently conceived, the inter-level message protocol supports two types of message which are designated 'D' and 'S', respectively. Table 3.1 shows the distinguishing characteristics of these two message protocols.

	Message Type 'D'	Message Type 'S'
Processing Order	arbitrary	FIFO
Length	fixed	variable

Table 3.1 - Characteristics of 'D' and 'S' Message Protocols

It is intended that 'D' format messages be used for bulk data transfers between levels, while 'S' format messages be used for passing service requests and control information between levels. The arbitrary processing order permitted for 'D' messages allows data blocks to be processed in a flexible order, while avoiding the overhead that would be incurred if incoming messages had to be physically removed from the incoming message queue in strict FIFO order. The variable length of 'S' messages is well suited for handling control information and service requests which come in various sizes and for which the fixed length 'D' protocol would be inappropriate. For 'S' messages, the limitation to FIFO processing is not a severe restriction since 'S' messages are expected to be fairly short, and thus the cost of physically moving them out of the incoming message queue should not be significant.

There is a separate sub-system within each GC for handling incoming 'S' and 'D' messages, respectively.

3.2.2.1 'S' Message Handling

A receiving GC places any incoming 'S' message in a circular buffer in LM called a Service Request Queue (SRQ). Acces to the SRQ is controlled via two registers in the GC, denoted SRQIN and SRQOUT, respectively. Register SRQIN contains the LM address of the next available location in the SRQ, and is updated by the GC whenever an 'S' message is stored into the SRQ. Register SRQOUT points to the first unprocessed message in the SRQ, and is updated by the LOS whenever a message is extracted from the SRQ. Figure 3.8 illustrates these operations. It can be seen that SRQIN and SRQOUT are continually

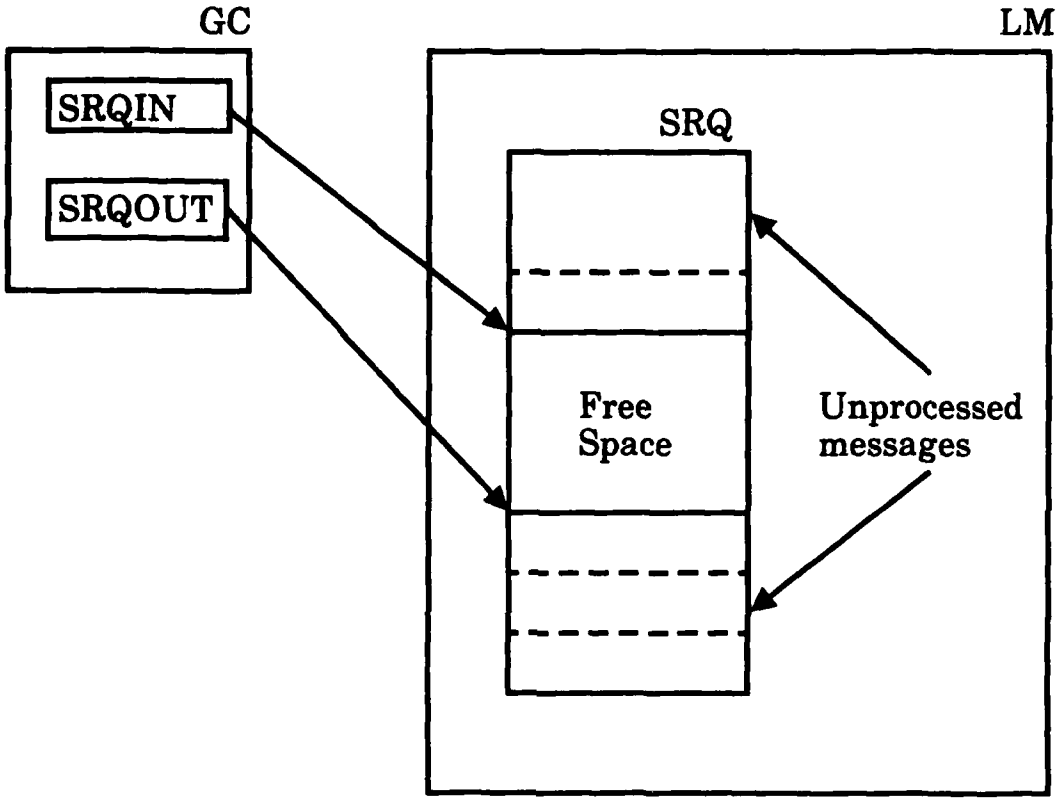


Figure 3.8--Illustration of SRQ Processing

"chasing" each other. If SRQIN catches up to SRQOUT, the SRQ is full, and the GC returns a busy signal to the sender. If SRQOUT catches up to SRQIN, it is an indication that the SRQ is empty.

3.2.2.2 'D' Message Handling

The handling of incoming 'D' messages is also based on a circular buffer in LM and a pair of registers in the GC. As before, the two registers, DRQIN and DRQOUT, are used to control access to the circular buffer which is designated the Data Request Queue (DRQ). In this case however, the DRQ contains addresses of data areas, rather than the data areas themselves, and the roles of the GC and the LOS in filling and emptying the circular are reversed. In other words, the LOS is responsible for keeping the DRQ filled with pointers to available (fixed size) data storage areas in LM. The GC extracts these pointers from the DRQ and uses them to place incoming data blocks in the appropriate place in LM. Management of the actual data storage areas must be handled by the LOS. All details of storage management are hidden from the GC, which sees only the DRQ.

As before, DRQIN and DRQOUT "chase" each other around the DRQ. DRQIN is incremented by the LOS after it places a new pointer into the DRQ at location DRQIN. If DRQIN catches up to DRQOUT, the DRQ is full. Similarly, DRQOUT is incremented by the GC after the pointer at location DRQOUT is used to place a data block in LM. If DRQOUT catches up to DRQIN, there is no available storage for the incoming data block, and the GC returns a busy signal to the sender. Figure 3.9 illustrates the operation of the 'D' protocol sub-system.

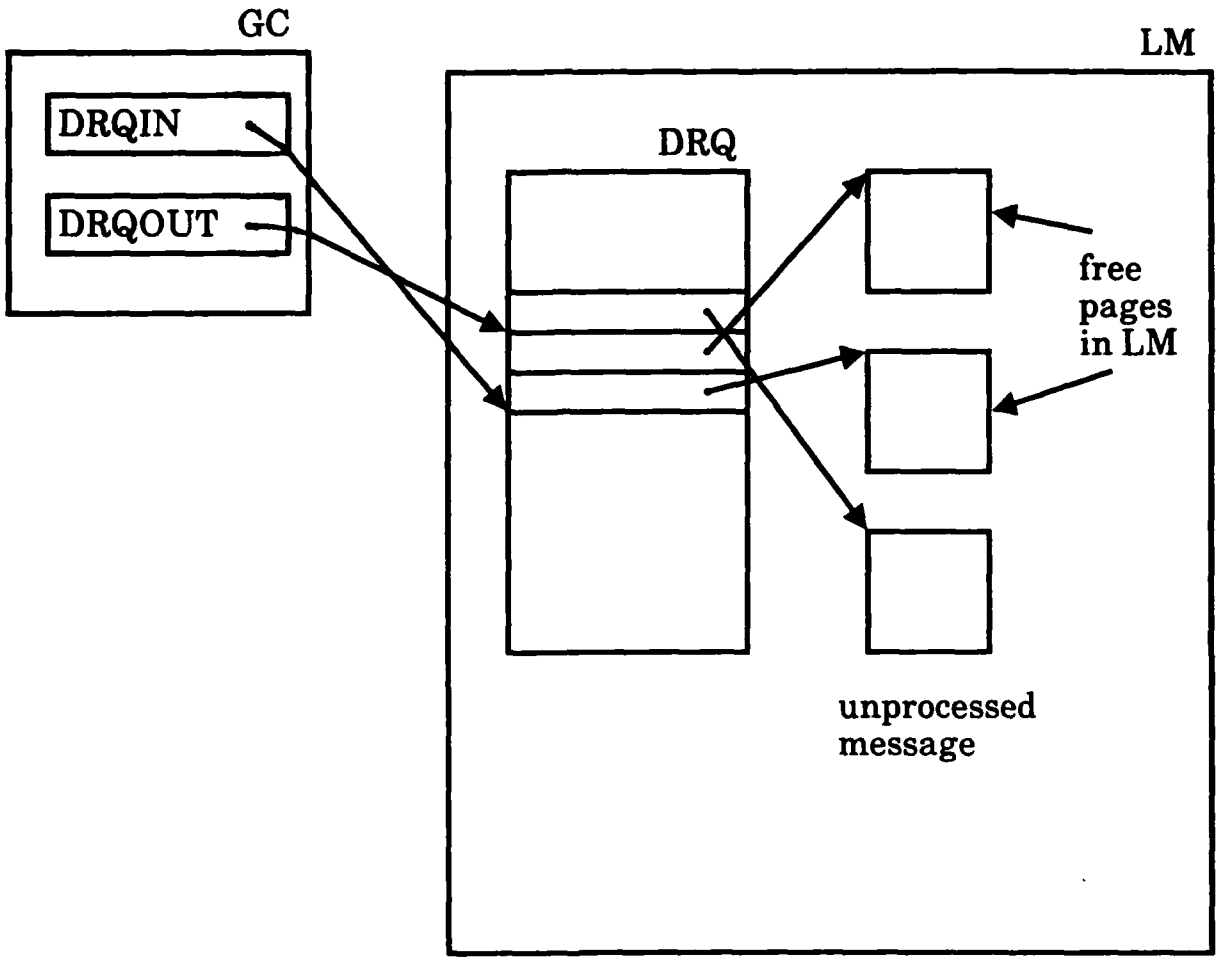


Figure 3.9--Illustration of DRQ Processing

One final piece of bookkeeping is needed. After the GC has finished filling a data storage block, it increments the DRQOUT register. It then generates an interrupt which signals the LOS that an incoming 'D' message has been completed. The LOS must then take note of the fact that an unprocessed message is located in LM at the address pointed to by DRQOUT-1.

3.3 Reliability Estimates for DSH-III

Due to the inherent redundancy of the multi-level structure of DSH-III, the system can be made extremely reliable by

a) replicating key components, such as the level 1 LBUS and the GBUS; and/or

b) employing a hardware/software strategy which enables the system to detect/diagnose failures and dynamically reconfigure itself to bypass failed components.

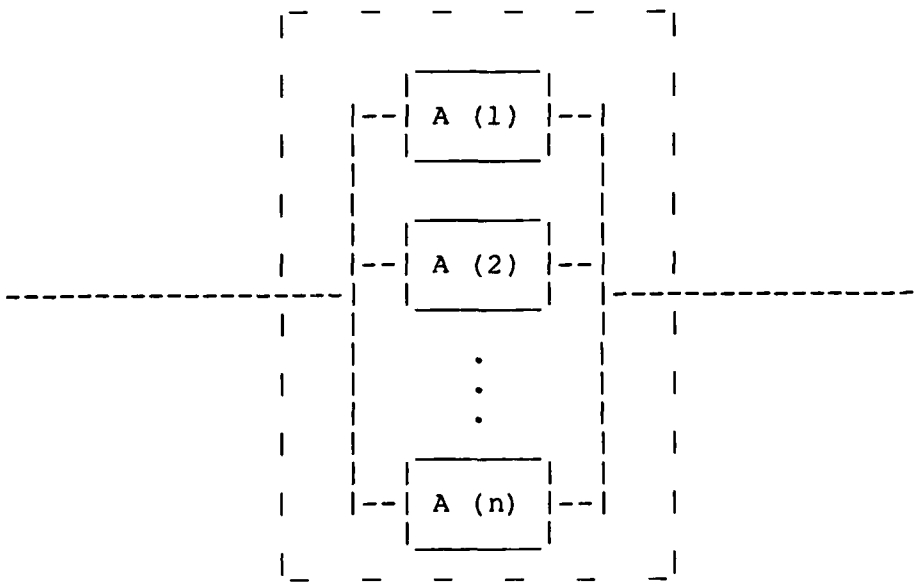
This strategy can have a dramatic effect on reliability, as is illustrated by the following development. (These calculations are very approximate, and are only intended to give order-of-magnitude estimates of the reliability attainable via various design choices.)

Define the Mean Time Between Failure (MTBF) for a component as the expected length of time between successive failures of that component. Define the Mean Time To Repair (MTTR) of a component as the average time needed to correct a failure in that component. Then we can define the Availability (A) of a component by

$$A = MTBF/(MTBF + MTTR) \qquad (1)$$

Intuitively, A is the fraction of time that the component is expected to be operational.

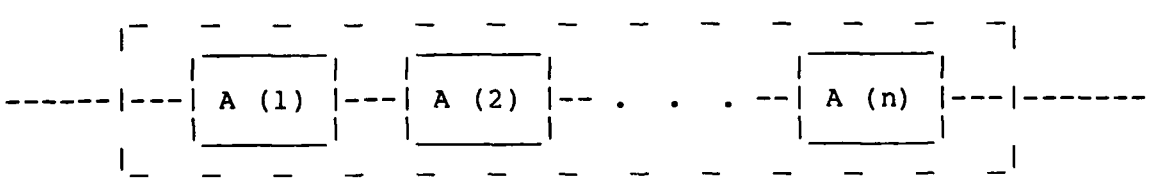
A system, S, composed of n redundant components with Availabilities given by A(1) to A(n) can be represented by



and its Availability, A(S), is given by

$$A(S) = 1 - \prod (1-A(i)) \qquad (2)$$

If system S is composed of n components, each of which must be working in order for S to be operational, then S can be represented by



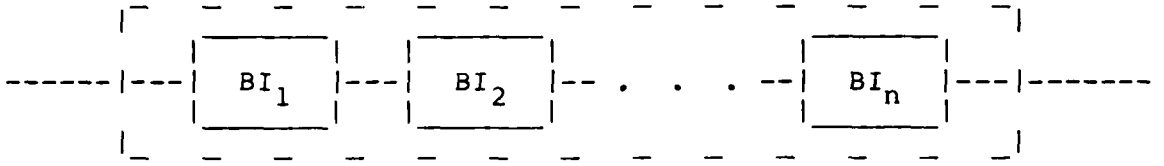
and the Availability, A(S), is given by

$$A(S) = \prod A(i) \qquad (3)$$

Now consider a system consisting of a single bus and some number, n, of Processing Elements (PE's). Each PE is interfaced to the bus by a Bus Interface (BI). To estimate the reliability of this system we make the following assumptions:

1) The bus itself will never fail.
2) If a BI fails it may hang the bus.
3) The system can continue to operate as long as the bus is operational. In this developemnt, we ignore the possibility of simultaneous failure of all n PE's, a very unlikely event if n is sufficiently large (>15).

Under these assumptions, the system will fail if a BI fails. (We assume, conservatively, that every BI failure will hang the bus.) The system can therefore be represented by



Here we are using the approximation that the failure of a single PE does not affect the operation of the system. Now, let the MTBF for each BI and PE be 10000 hours, let the MTTR for a BI be 24 hours, and suppose that n=30. Then the Availability of this system can be obtained from Eqs. 1 and 3 as

$$A(BI) = 10000/(10000+24) = .9976$$

$$A(S) = A(BI)^{30} = .9976^{30} = .9305$$

The system MTBF, MTBF(S), can now be computed by inverting (1) to give

$$MTBF = A*MTTR/(1-A) \quad (4)$$

which leads to

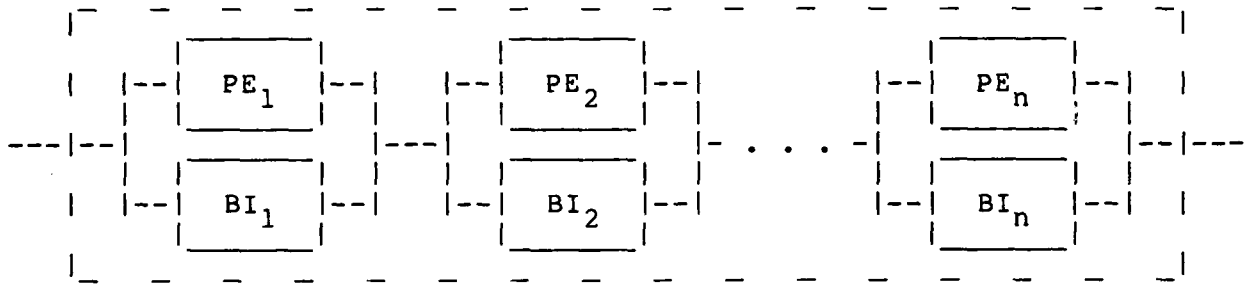
$$MTBF(S) = A(S)*MTTR/(1-A(S)) = .9305*24/(1-.9305) = 321.32 \text{ hours.}$$

In this case, redundancy has led to a lower system reliability than the individual component reliabilities by increasing number of ways in which the system can fail.

Now introduce an additional assumption

4) Firmware within each processor can be used to diagnose and "amputate" failed BI's, thus freeing the bus.

Under this further assumption, the system can be represented by



since the bus is disabled only if a BI and its corresponding PE fail simultaneously. Under this set of assumptions, we get (using Eqs. 1 - 4)

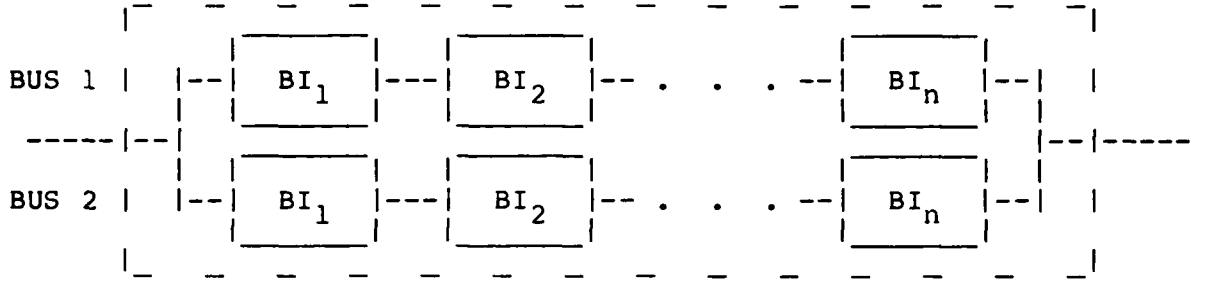
$A(S) = .9998$, and

$MTBF(S) = 119{,}979$ hours.

Now consider modifying the structure of the system by replicating the bus. Replace assumption 4 by

4) If a BI fails and hangs a bus, the system can continue operation using the other bus.

This assumption leads to a system represented by



since the system fails only if at least one BI on each bus fails. This system has

A(S) = .9952

MTBF(S) = 4,976 hours.

If this system is enhanced by incorporating logic which allows the system to perform self-diagnosis and reconfiguration as long as at least one bus is working, then the effective MTTR for the failed bus is greatly reduced. In other words, if the system can fairly rapidly restore a hung bus to normal operation, the "window" during which multiple bus failures can occur is greatly reduced. If we assume that the system can amputate failed components within .1 hours, we get

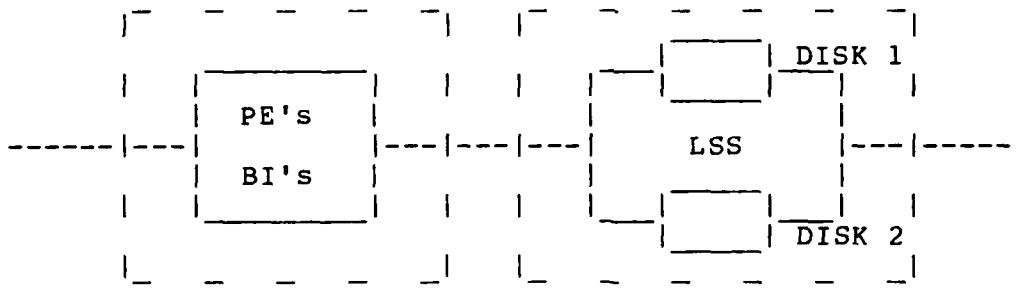
A(S) = .999999

MTBF(S) = $2.4*10^6$ hours.

This analysis illustrates the fact that a highly reliable system can be built using only double redundancy. Of course, this analysis takes advantage of the fact that there is multiple redundancy of BI's and

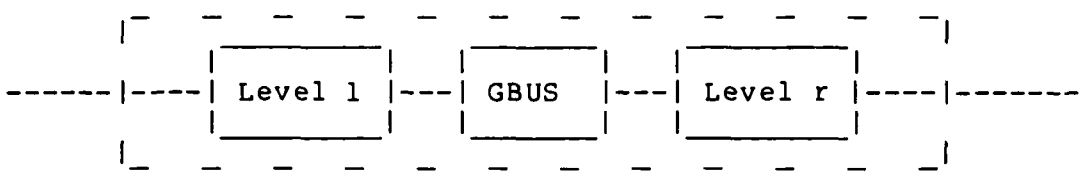
PE's. We are making the implicit assumption that a system with 30 PE's is essentially equivalent to a system with 29 PE's. This assumption justifies our treatment of amputation as restoring a system to normal operation.

The system just analyzed corresponds closely to Level 1 of DSH-III. A similar analysis can be performed for other key DSH-III sub-systems, the reservoir and the GBUS. Analysis of the reservoir is similar to that of level 1 except that an LSS must be included. Assume for simplicitly that the LSS is composed of two independent redundant disk subsystems, each with an MTBF of 5000 hours. Then the reservoir can be represented by



This system has an MTBF of approximately 1 million hours.

Assuming that the GBUS can be made at least as reliable as the reservoir allows the entire DSH-III system to be represented by



This gives a system MTBF of over a quarter of a million hours.

4 ALGORITHMS TO SUPPORT THE READ OPERATION

This section describes the transaction protocols, software algorithms, and major data structures necessary to implement the READ algorithm presented in Section 2 in an environment such as the one provided by the hardware and software architecture presented in Section 3.

The organization of this section is as follows. First, we provide an overview of the READ algorithm together with a number of examples which illustrate major features of the implementation. Next, the key data structures used by the algorithms at each level are described. Finally, detailed descriptions of the READ algorithms are presented.

Recall, from Section 2, that the READ algorithm implements a GLOBAL-LRU-SOP data management policy, and, therefore, embodies the following properties:

- MLOI is automatically maintained at all levels of the storage hierarchy;
- overflows can be ignored, since Static Overflow Placement (SOP) reduces to a vacuous operation if the MLOI property holds.

In order to obtain these properties, the READ algorithm must observe a number of constraints, including:

- the number of blocks in each level must be strictly greater than the number of blocks in the next higher level, i.e., we must have $m^i > m^{i-1}$, $i = 2,\ldots,r$, as prescribed by Theorem 3 of Section 2;
- pages must be selected for eviction from each level using a Least Recently Used (LRU) replacement policy;
- each level must perform exactly the same sequence of LRU stack updates.

4.1 Overview and Examples

In Section 2, the overall functioning of a READ operation was described. Briefly, the READ request is percolated downward through the hierarchy until it reaches a level containing the requested data. This data is then broadcast to all higher levels, and passed back from Level 1 to the user.

The algorithms at each level are organized around three major data structures: The LSS, the directory, and the pending request queue (PRQ).

The LSS contains that subset of the DSH-III virtual address space currently residing in the level. Recall that the LSS may consist of a disk storage sub-system, may actually reside in LM at the level, or may be implemented by some other storage technology. However the LSS is actually implemented, for the purposes of this discussion it is viewed as a black box which is accessible via the following primitives (or their equivalents):

- STORE(data area specification,LSS id) - instructs LSS to store the data block specified by data area specification and to return LSS id which is an identifier for the data block which is used for subsequent LSS accesses to the data.

- RETRIEVE(data area specification,LSS id) - instructs LSS to retrieve the data block specified by LSS id and place it in LM at the location specified by data area specification.

- DELETE(LSS id) - instructs LSS to delete the data block specified by LSS id, thus freeing its storage for reuse.

- UPDATE(data area specification,LSS id) - instructs LSS to update the data block specified by LSS_id with the data specified by data area specification.

The directory for a level consists of two components. The first component is a data table which contains an entry for every data block which is either stored at a level or in the process of being retrieved

into the level. Entries for stored data blocks are chained together on a bi-directional linked list in LRU order. In addition, each data table entry contains a number of status and control fields, e.g., the LSS_id for the block. The second component of the directory is a hashed scatter table which contains pointers to data table entries, and is used to access individual data table entries.

The pending request queue (PRQ) is a per-block list of READ requests for a block which have yet to be satisfied at a level. The major function of this list is to record duplicate requests for the same data block. For example, requests for Xa and Xb will both result in references to X* at Level 1. If X* is not in Level 1, one READ, for Xa say, will be passed on down to Level 2, and the other READ, for Xb, will be saved on the Level 1 PRQ for X*. When X* is eventually broadcast to Level 1 from below, the PRQ for X* will be processed and Xb will be returned to the requesting user. Use of the PRQ in this case has eliminated a duplicate request for X* at Level 2. The examples which follow show that use of the PRQ also eliminates duplicate LSS requests and, in addition, allows all requests to be handled in a uniform manner, thus simplifying the algorithms.

Detailed descriptions of the formats of the directory and the PRQ are given in section 4.2.

We now present three examples, in order of increasing complexity, which illustrate some of the foregoing ideas, and serve as an introduction to the detailed algorithm descriptions which follow.

4.1.1 Example 1: READ With Data Found in Level 1

This example illustrates the simplest possible READ. A user issues a request to READ data block Xa, and its parent block, X*, is present in Level 1. Figure 4.1 shows the processing steps which are detailed below.

(1) User issues READ(request id,Xa), and a READ transaction is sent to Level 1 via the UBUS.

(2) The directory at Level 1 is searched, and data block X* is located. The LSS_id for X*, LSS_id.X*, is extracted from the directory.

(3) The request for Xa is put on the PRQ for X*.

(4) RETRIEVE(LSS_id.X*) is sent to the LSS.

(5) The LSS returns the requested data to the LM via an LSS_DATA transaction.

(6) The arrival of X* in LM triggers the processing of the PRQ for X*. In this case, the only entry on the PRQ for X* is the READ request for Xa.

(7) A DATA(request id,Xa) transaction is sent to the requesting user.

This example, and the two following, ignore the issue of LRU stack handling. This issue is dealt with in the detailed algorithm specifications in Section 4.3.

4.1.2 Example 2: READ With Data Found in Level 3

This example illustrates the percolation of a READ down through the hierarchy until the requested data is located. In this case, the highest level containing the requested data, block WXYZ, is Level 3. Figure 4.2 illustrates the processing steps which are detailed below.

(1) READ(request id,WXYZ) sent to Level 1 via the UBUS.

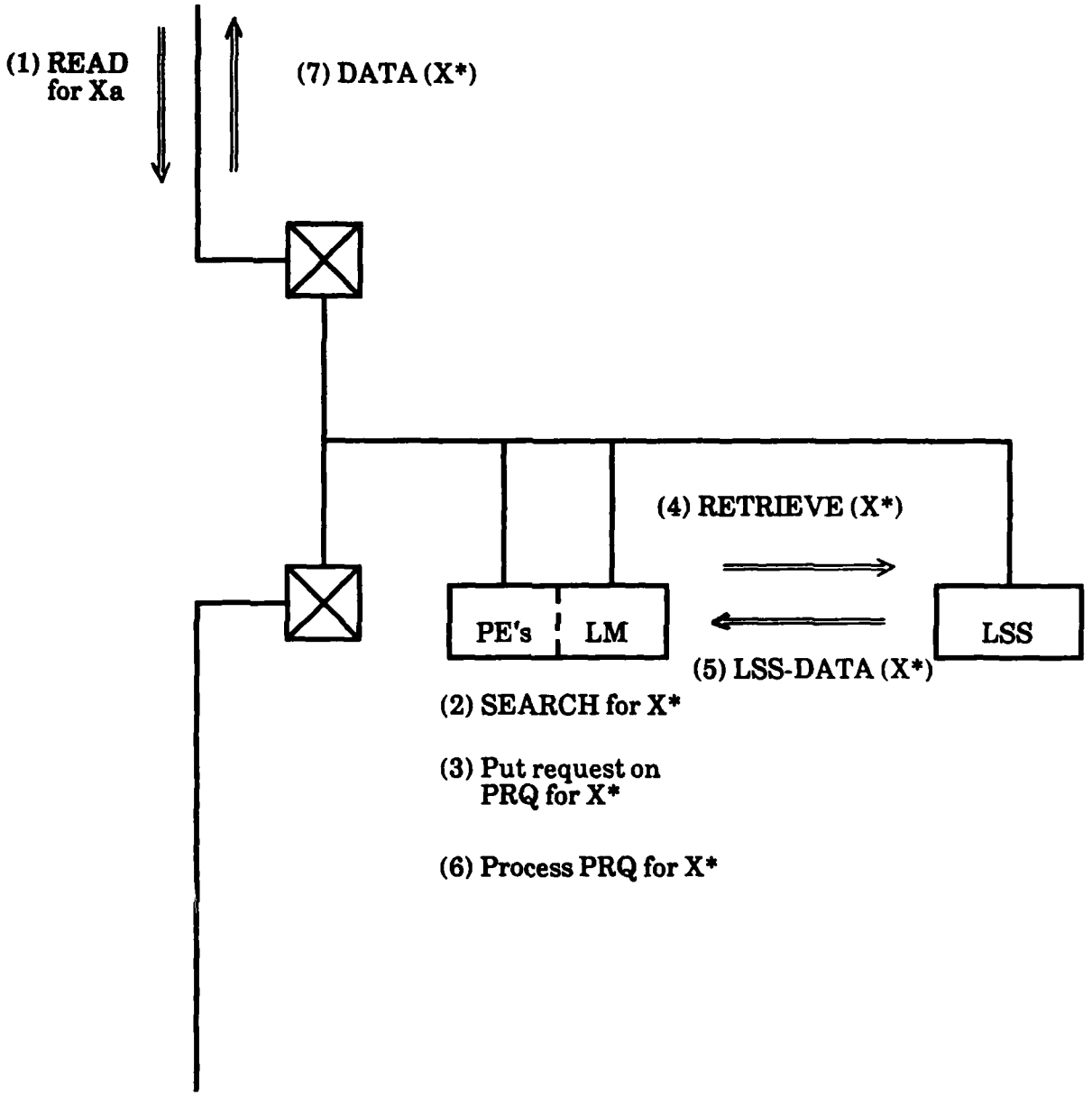


Figure 4.1--Example of READ

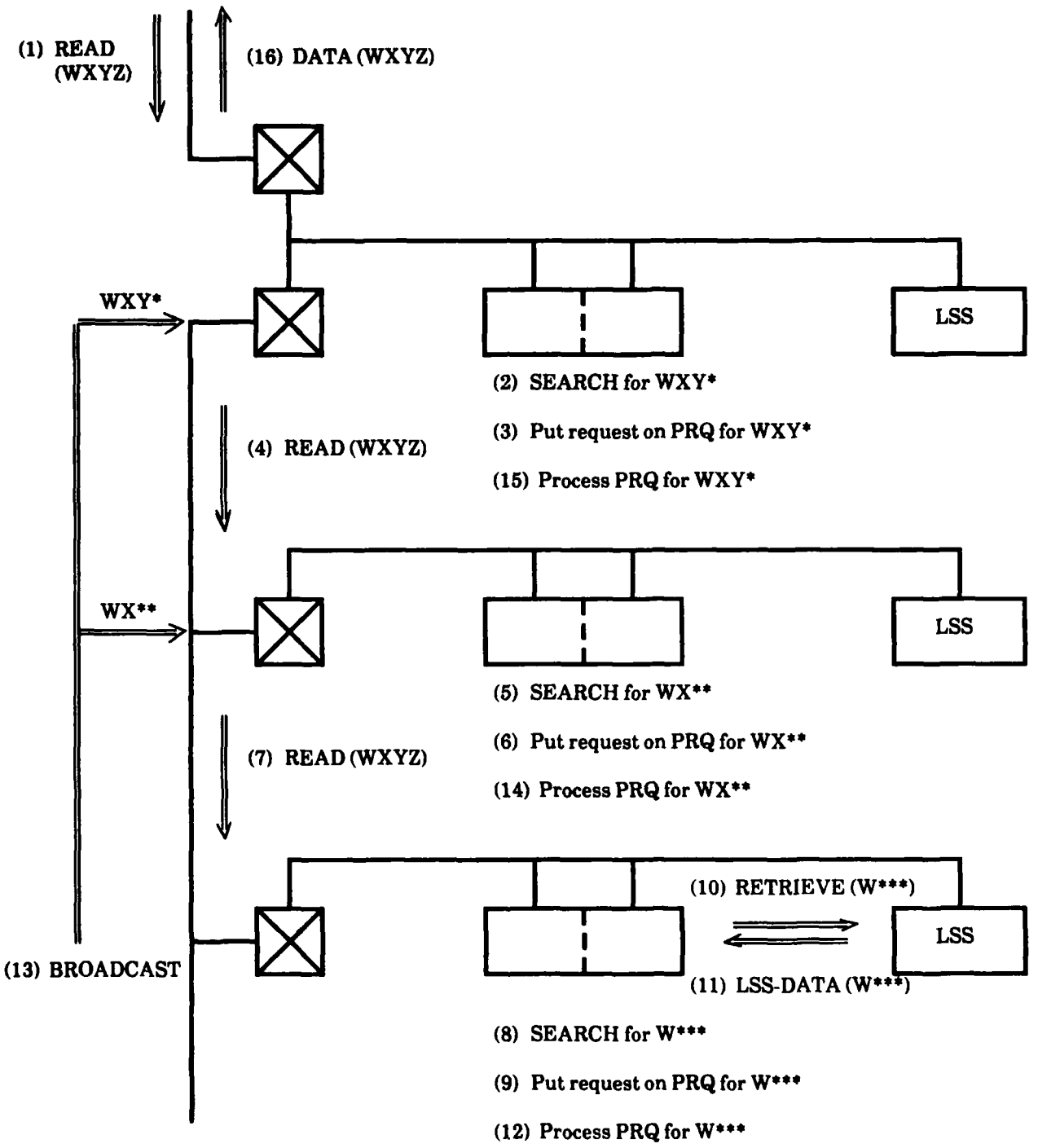


Figure 4.2--Example of a READ

(2) The Level 1 directory is searched for WXY*, which is not found.

(3) The READ request for WXYZ is put on the PRQ for WXY*.

(4) READ(request id,WXYZ) is sent to Level 2 via the GBUS.

(5) The Level 2 directory is searched for WX**, which is not found.

(6) The READ request for WXYZ is put on the PRQ for WX**.

(7) READ(request id,WXYZ) is sent to Level 3 via the GBUS.

(8) The Level 3 directory is searched for W*** which is found. The LSS_id for W***, LSS_id.W***, is extracted from the directory.

(9) The READ request for WXYZ is put on the PRQ for W***.

(10) RETRIEVE(LSS_id.W***) is sent to the LSS.

(11) The LSS places the requested data in the LM via a LSS_DATA transaction.

(12) The PRQ for W*** is processed. In this case, the request which was saved in (9) triggers a broadcast to all higher levels.

(13) The requested data is broadcast to Levels 1 and 2. The data is broadcast as follows:

Data	Destination
WX**	Level 2
WXY*	Level 1

(14) When WX** arrives at Level 2, the PRQ for WX** is processed. In this case, since the broadcast has accomplished the goal of sending WXY* to Level 1, nothing further need be done, and the READ request for WXYZ on the PRQ is discarded.

(15) When WXY* arrives at Level 1, the PRQ for WXY* is processed. The only request on the queue is the READ for WXYZ.

(16) DATA(request id,WXYZ) is sent to the requesting user.

For the sake of clarity, this example has glossed over the details of data block eviction and LSS storage of incoming data. The reader can think of these functions as being handled by background processes asynchronously with the rest of the processing.

4.1.3 Example 3: Simultaneous READS for the Same Data Block

This example illustrates the major function of the PRQ, namely the elimination of duplicate READs for the same data. Briefly, two READs, for XYa and XZb, are issued close together in time. The common parent for the requested data blocks, X**, is not currently in Level 2, and so must be retrieved from lower down the hierarchy. The algorithm operates in such a fashion that only one of the two original READs triggers a search below Level 2. The other READ is saved on the appropriate PRQ at Level 2, and is finally processed when X** arrives at Level 2 as a result of the first READ. Figure 4.3 illustrates the processing steps for the two READs. The numbering of the steps is somewhat arbitrary, since much of the processing can be done in parallel. In the explantion below, a parenthesized number after the description of a step indicates a prerequisite step.

(1) READ for XYa is issued.

(2) READ for XZb is issued.

(3) Level 1 directory is searched for XY*, which is not found. (1)

(4) READ for XYa is put on PRQ for XY*. (3)

(5) READ for XYa is sent to Level 2. (4)

(6) Level 2 directory is searched for X**, which is not found. (5)

(7) READ for XYa is put on PRQ for X**. (6)

(8) READ for XYa is sent to Level 3. (7)

(9) The hierarchy below Level 2 is searched for a block containing X**. Eventually this block is located, and a READ THROUGH is initiated in Step 15. (8)

(10) Level 1 directory is searched for XZ*, which is not found. (2)

(11) READ for XZb is put on PRQ for XZ*. (10)

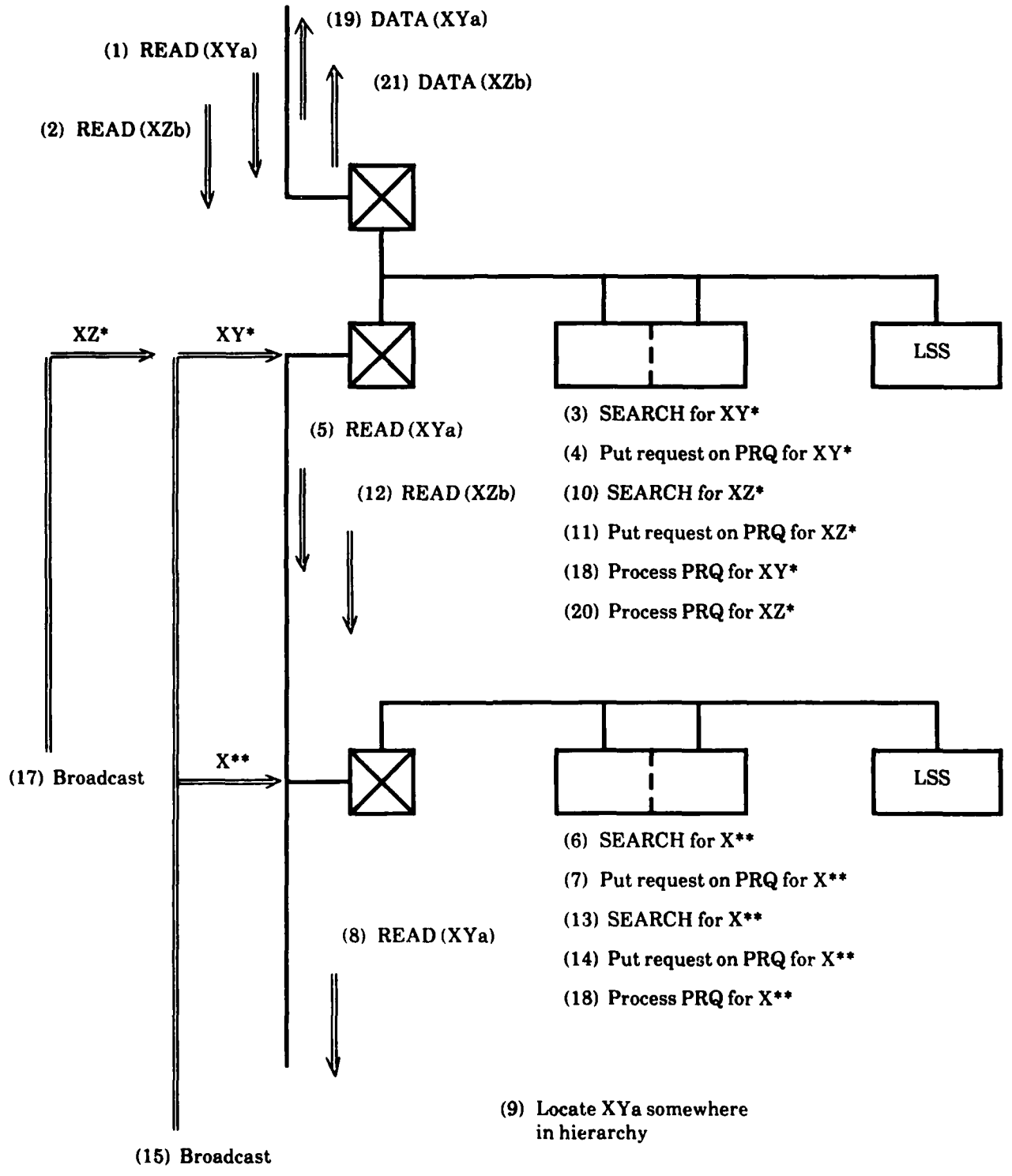


Figure 4.3 Example of Two Simultaneous READs

(12) READ for XZb is sent to Level 2. (11)

(13) Level 2 directory is searched for X**. This block is not found but the pending read for X** (put on the PRQ in Step 7) is noted. (12)

(14) READ for XZb is added to the PRQ for X**. Since there is already an outstanding READ for X**, the request for XZb is held until X** arrives at Level 2. (13)

(15) The request for XYa sent to Level 3 in Step 8 is finally satisfied, and the requested data is broadcast as follows:

Data	Destination
X**	Level 2
XY*	Level 1

(16) When X** arrives at Level 2, the PRQ for X** is processed. There are two requests on the queue, the first for XYa and the second for XZb. Since the broadcast of Step 15 has satisfied the first request, it is discarded. The second request results in a broadcast of XZ* to Level 1. (15)

(17) XZ* is broadcast to Level 1. (16)

(18) When the broadcast generated in Step 15 arrives at Level 1, the PRQ for XY* is processed. The only request on the queue is the READ for XYa. (15)

(19) XYa is sent to the requesting user. (18)

(20) When the broadcast generated in Step 17 arrives at Level 1, the PRQ for XZ* is processed. The only request on the queue is the READ for XZb. (17)

(21) XZb is sent to the requesting user. (20)

4.2 Data Structure Formats

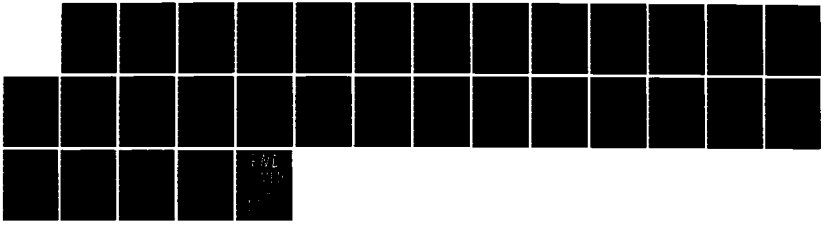
This section describes the formats of the key data structures used by the READ algorithms of DSH-III.

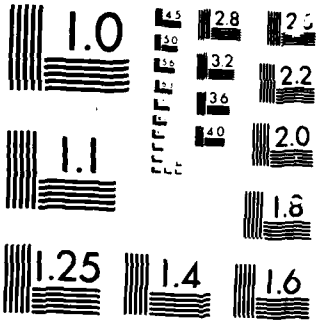
4.2.1 Directory Format

The directory at each level consists of a data table containing a fixed number of information entries ("slots"), and a scatter table which allows hashed access to the data table. Hash collisions are

AD-A166 287 DESIGN OF A DATA STORAGE HIERARCHY DSH-III--SOFTWARE & HARDWARE(U) ALFRED P SLOAN SCHOOL OF MANAGEMENT CAMBRIDGE MA M J ABRAHAM MAR 86 M010-8603-19 2/2

UNCLASSIFIED N00039-83-C-0463 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART

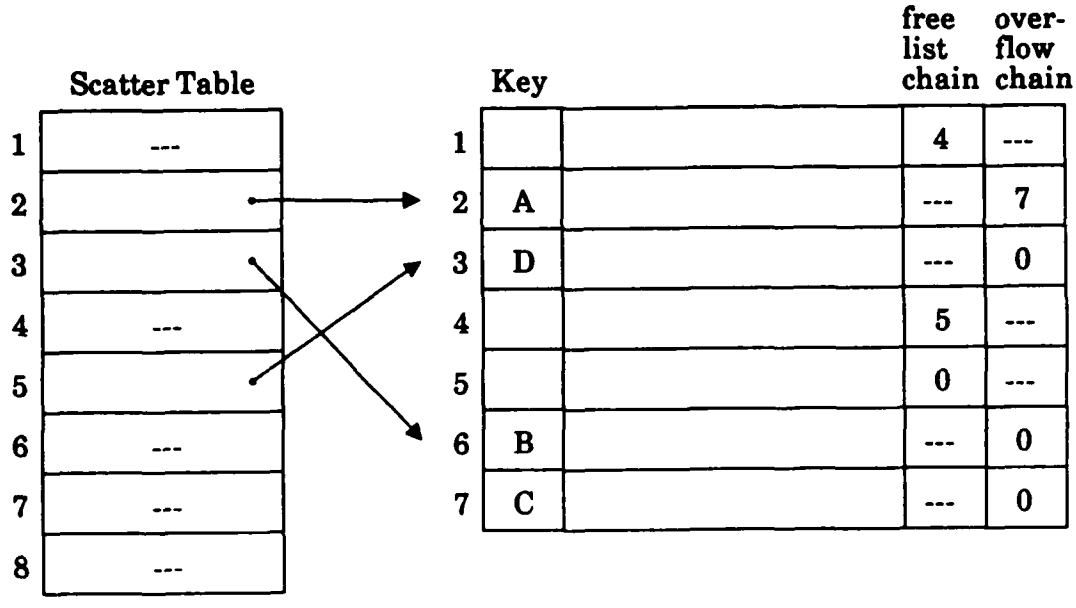
handled via overflow chaining. This policy has been selected since it performs better than other policies, such as open addressing, in situations where the majority of references are successful searches []. Figure 4.4 shows a typical directory. Note that unused ("free") data table slots are maintained on a free chain. We show the logic for the SEARCH, ADD, and DELETE functions to show how this data structure is managed.

The logic descriptions which follow are presented in a "pseudo-PL/I" format. Program variables (e.g., the fields in a directory entry) are all lower case. The keywords of "pseudo-PL/I" have initial upper case letters (e.g., Select). Subroutine names are all upper case (e.g., SEARCH).

4.2.1.1 Logic for SEARCH Function

SEARCH is invoked by SEARCH(key,slot), where virt_addr is the search key of the item to be located and slot is the data table entry number of the item (returned by SEARCH). SEARCH returns the string 'FOUND' or 'NOTFOUND' to its point of invocation, as appropriate.

```
return_val = 'NOTFOUND';                    /* assume item not in table
searchloop:
Do index = scatter_table(HASH(virt_addr)) /* loop over overflow chain
           Repeat data_table(index).overflow
           While (index ^= 0);
    If data_table(index).key = virt_addr  /* is this the one?
    Then Do;                              /* yes, it is
        slot = index;
        return_val = 'FOUND';
        Leave searchloop;
        End;
    End;
Return ( return_val );
```



Hash Function:	Key	H(Key)
	A	2
	B	3
	C	2
	D	5

Free chain anchor: 1

Figure 4.4 Directory Structure

4.2.1.2 Logic for ADD Function

ADD is invoked by ADD(virt_addr, slot) where the arguments have the same meaning as for SEARCH. For the sake of simplicity, this version of ADD assumes that the item to be added does not already exist and that at least one free slot is available.

```
slot = free_chain_anchor;                    /* get slot off free chain
free_chain_anchor = data_table(slot).free_chain_index;
                                             /* pop new slot off chain

index = HASH(virt_addr);                     /* compute scatter table index

If scatter_table(index) = 0                  /* check for collision
Then Do;                                     /* no collision
    scatter_table(index) = slot;             /* point scatter table at slot
    data_table(slot).overflow = 0;           /* initialize overflow chain
    End;
Else Do;                                     /* a collision
    Do index = scatter_table(index)          /* loop over overflow chain
               Repeat data_table(index).overflow
               While (data_table(index).overflow ^= 0);
        End;
    data_table(index).overflow = slot;
                                             /* point to new entry
    data_table(slot).overflow = 0;           /* initialize overflow chain
    End;
```

4.2.1.3 Logic for DELETE Function

DELETE is invoked by DELETE(slot), where slot is the data table entry number for the item to be deleted. For simplicity, we assume that slot refers to a non-free entry.

```
key = data_table(slot).key;                    /* get item key from table
scat_index = HASH(key);                        /* get scatter table index
index = scatter_table(scat_index);             /* get index of first item
                                               /* on chain
If index = slot                                /* is this the one?
Then Do;
    scatter_table(scat_index) = 0;             /* just erase scatter entry
    End;
Else Do;
    Do index = index
               Repeat data_table(index).overflow
               While (data_table(index).overflow ^= slot);
        End;
    data_table(index).overflow = data_table(slot).overflow;
    End;

data_table(slot).free_chain_index = free_chain_anchor;
                                               /* put slot on free chain
free_chain_anchor = slot;
```

4.2.2 LRU Chains and PRQ Format

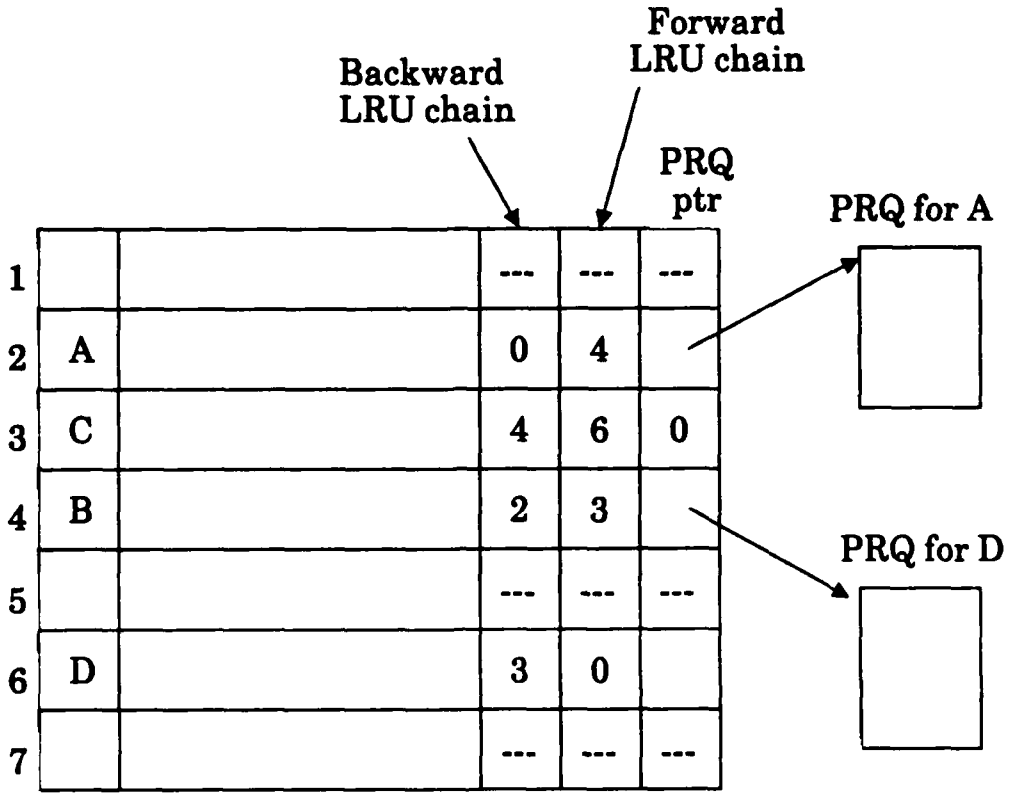
LRU processing is handled via a bi-directional chain running through the slots in LRU order. The PRQ for a block is accessed via a pointer in the data table slot for that block. Figure 4.5 illustrates these pointers.

Each PRQ is implemented as a linked list which is anchored by the pointer in the corresponding data table slot.

4.3 READ Algorithms and Transactions

This section presents a fairly detailed description of the algorithms and transactions supporting the READ operation. However, there are a number of implementation issues which are somewhat peripheral to the logical structure of the algorithms; these issues are not dealt with here. They include

- precise specification of LSS operation for various classes of LSS technology;



LRU order is: A, B, C, D

Figure 4.5 Illustration of LRU & PRQ Pointers

- details of LM management in general, and buffer management in particular;

- precise specification of inter-level message handling including packet assembly/disassembly and SRQ/DRQ management.

This section does describe management of the key data structures, the directory and the PRQ, and gives a logically complete description of the data flow for READs within DSH-III.

The transactions on which the READ algorithms are based are summarized in Table 4.1. The FORMAT column of Table 4.1 shows the message protocol used for each transaction type. Possible values of FORMAT are: 'D', used for inter-level data transmission; 'S', used for inter-level control and service request messages; and 'I', used for intra-level messages.

The arrival of any one of these transaction types triggers the activation of an appropriate process to handle the transaction. Thus, a READ transaction triggers the READ process. In addition, there are some background processes which perform tasks that are not keyed to transaction arrivals. For example, the EVICT process attempts to maintain the number of free blocks at a prespecified level asynchronously with the reuse of these blocks by other processes. Table 4.2 summarizes some of the utility subroutines invoked by the transaction handling processes.

There are also a number of low-level utility subroutines used by the READ algorithms. These subroutines are summarized in Table 4.3.

The following sections describe the transaction processes under the assumption that they are running in a uni-processor environment at each level. That is, concurrent update issues are ignored. There are two related problems in this area. Shared data must be locked to

Transaction/ Arguments	Format	Description
READ	S	sent by user to Level 1, or by a level to next lower level, requesting data
req_id		user assigned request identifier
virt_addr		virtual address of requested data block
LSS_DATA	I	sent by LSS to LOS within a level; contains data requested from LSS by previous RETRIEVE transaction
lm_ptr		pointer to LM location of retrieved data block
slot		directory slot number of data block
BCAST_DATA	D	broadcast by a level to all higher levels in response to READ request
lm_ptr		pointer to LM location of data block
virt_addr		virtual address of data block
DATA	D	sent by level 1 to a user
req_id		user supplied identifier for request
data		data block requested by READ request
NOTIFY	I	intra-level message signalling the availability of a previously requested data block
slot		directory slot number of data block
STORE	I	sent to LSS to request storage of a data block
lm_ptr		LM location of data block to be stored
slot		directory slot number of data block
STORE_ACK	I	sent to LOS by LSS acknowledging STORE request
lss_id		LSS assigned identifier for stored data block
slot		directory slot number of data block
RETRIEVE	I	sent to LSS to request retrieval of previously stored data block
lss_id		LSS identifier of previously stored block
LRU_UPDATE	S	request for LRU stack update broadcast to all levels simultaneously
virt_addr		virtual address of block for which LRU update is to be performed

Table 4.1 - Summary of Transactions Used by READ

prevent update inconsistencies and the locking of shared data could cause bottlenecks due to contention. Later in this chapter, we

Routine Name/ Arguments	Description
SEARCH	performs directory lookup; invoked as a function, returns 'FOUND' or 'NOTFOUND' as appropriate
virt_addr	virtual address of data block
lss_id	LSS assigned identifier for data block
slot	directory slot number for data block
DIR_ADD	adds a directory entry
virt_addr	virtual address of entry to be added
slot	directory slot number of new entry (output)
PRQ_ADD	adds an entry to the PRQ for a block
virt_addr	virtual address of block
req_id	request identifier for request to be added
slot	directory slot number of data block
addr	address of update data for WRITE NULL pointer for READ
PRQ_POP	pops entries off PRQ in FIFO order; invoked as a function, returns 'EMPTY' if the PRQ is empty
slot	directory slot number for PRQ
virt_addr	virtual address of popped request
req_id	request identifier of popped request
DO_LRU	performs an *LRU stack update*
slot	slot number to be moved to top of stack

Table 4.2 - Summary of Functional Routines Used by READ Transactions

indicate the modifications that would be necessary to permit these programs to run in a multi-processor environment. These modifications address the locking problem and minimize the possibility of bottlenecks by segmenting the address space at each level, thus allowing n-fold overlapped processing for n independent sets of system tables at each level.

Routine Name/ Arguments	Description
LEVEL	returns the DSH-III level of its caller
FREE ptr	releases a block of LM storage address of block to be freed
RECYCLE	relinquishes control to allow higher priority tasks to run
The following routines generate destination addresses for message sending subroutines.	
ALL	returns a destination list containing all levels
USER	returns the address of the user of DSH-III
HIGHER level	returns a destination list containing all levels above its argument
LSS level	returns address of LSS at level
LOS level	returns address of LOS at level

Table 4.3 - Summary of Utility Routines Used by READ Transactions

4.3.1 The READ Transaction - READ(req_id, virt_addr)

```
Select( SEARCH(virt_addr, lss_id, slot) ); /* directory lookup

When( 'NOTFOUND' ) Do;                     /* block not in level
    DIR_ADD(virt_addr, slot);              /* create directory entry
    dir(slot).status = 'PENDING';          /* note waiting for block
    PRQ_ADD(virt_addr, req_id, slot, NULL);/* remember request
    READ(req_id, virt_addr, LEVEL+1);      /* forward READ downwards
    End /* of handling data not found in level */;
```

```
When( 'FOUND' ) Do;                              /* block in directory
    PRQ_ADD(virt_addr, req_id, slot, NULL);/* remember request
    Select( dir(slot).status );                  /* branch on status
    When( 'PENDING' );                           /* already waiting
    When( 'HOLD' );                              /* data already in LM
    When( 'INLSS' ) Do;                          /* data is in LSS
        dir(slot).status = 'PENDING';            /* note waiting for block
        RETRIEVE(lss_id, slot, LSS(LEVEL));/* ask LSS for data block
        End /* of INLSS processing */;
    Otherwise Error;                             /* illegal status
    End /* of status selection */;
    End /* of handling data found in level */;

Otherwise Error;                                 /* bad Search return code

End /* of READ processing */;
```

4.3.2 The LSS_DATA Transaction - LSS_DATA(lm_ptr, slot)

```
dir(slot).lm_ptr = lm_ptr;                       /* save LM location of block
NOTIFY(slot, LOS(LEVEL));                        /* alert arrival of data
```

4.3.3 The BCAST_DATA Transaction - BCAST_DATA(lm_addr, virt_addr)

```
If SEARCH(virt_addr, lss_id, slot) ^= 'FOUND' Then Error;
                                                 /* get slot number
dir(slot).lm_ptr = lm_addr;                      /* save LM pointer
If LEVEL ^= 1 Then PRQ_POP(virt_addr, req_id, slot, type);
                                                 /* discard first request
                                                 /* since broadcast sends
                                                 /* data to upper levels
STORE(dir(slot).lm_ptr, slot, LSS(LEVEL)); /* send data to LSS
```

4.3.4 The NOTIFY Transaction - NOTIFY(slot)

```
dir(slot).status = 'HOLD';                       /* lock the block

Do While( PRQ_POP(virt_addr, req_id, slot, type) ^= 'EMPTY' );
    Select( LEVEL );                             /* branch on level
    When( 1 ) Do;                                /* if level 1 ...
        LRU_UPDATE(virt_addr, ALL);              /* send LRU update to all
                                                 /* levels
        DATA(dir(slot).lm_ptr, virt_addr, USER);
                                                 /* send data to user
        End /* of level 1 processing */;
```

```
    Otherwise Do;                              /* if not level 1 ...
        DATA(dir(slot).lm_ptr, virt_addr, HIGHER(LEVEL));
                                               /* broadcast data to all
                                               /* higher levels
        End /* of processing for all levels except 1 */;
    End /* of level selection */;
    RECYCLE;                                   /* redispatch to enhance
                                               /* interleaving
End /* of loop over all requests on PRQ */;

dir.status(slot) = 'INLSS';                    /* note block is in LSS
FREE(dir(slot).lm_ptr;                         /* free block's LM storage
```

4.3.5 The LRU_UPDATE Transaction - LRU_UPDATE(virt_addr)

```
If SEARCH(virt_addr, lss_id, slot) ^= 'FOUND' Then Error;
DO_LRU(slot);                                  /* perform LRU update
```

4.3.6 The STORE_ACK Transaction - STORE_ACK(lss_is,slot)

```
dir(slot).lss_id = lss_id;                      /* save LSS identifier
NOTIFY(slot, LOS(LEVEL));                       /* awake PRQ process
```

4.3.7 The EVICT Process

At any point in time, three classes of slots may exist in a directory. These classes have been mentioned briefly above. The classes are

free - slots currently on the free chain;

stored - slots currently assigned to data blocks stored at the level, and on the LRU chain;

pending - slots to which a pending data block has been assigned, but which have not yet been added to the LRU stack.

These classes are mutually exclusive and collectively exhaustive. Figure 4.6 shows a state transition diagram which illustrates how various transactions cause a slot to move from one class to another.

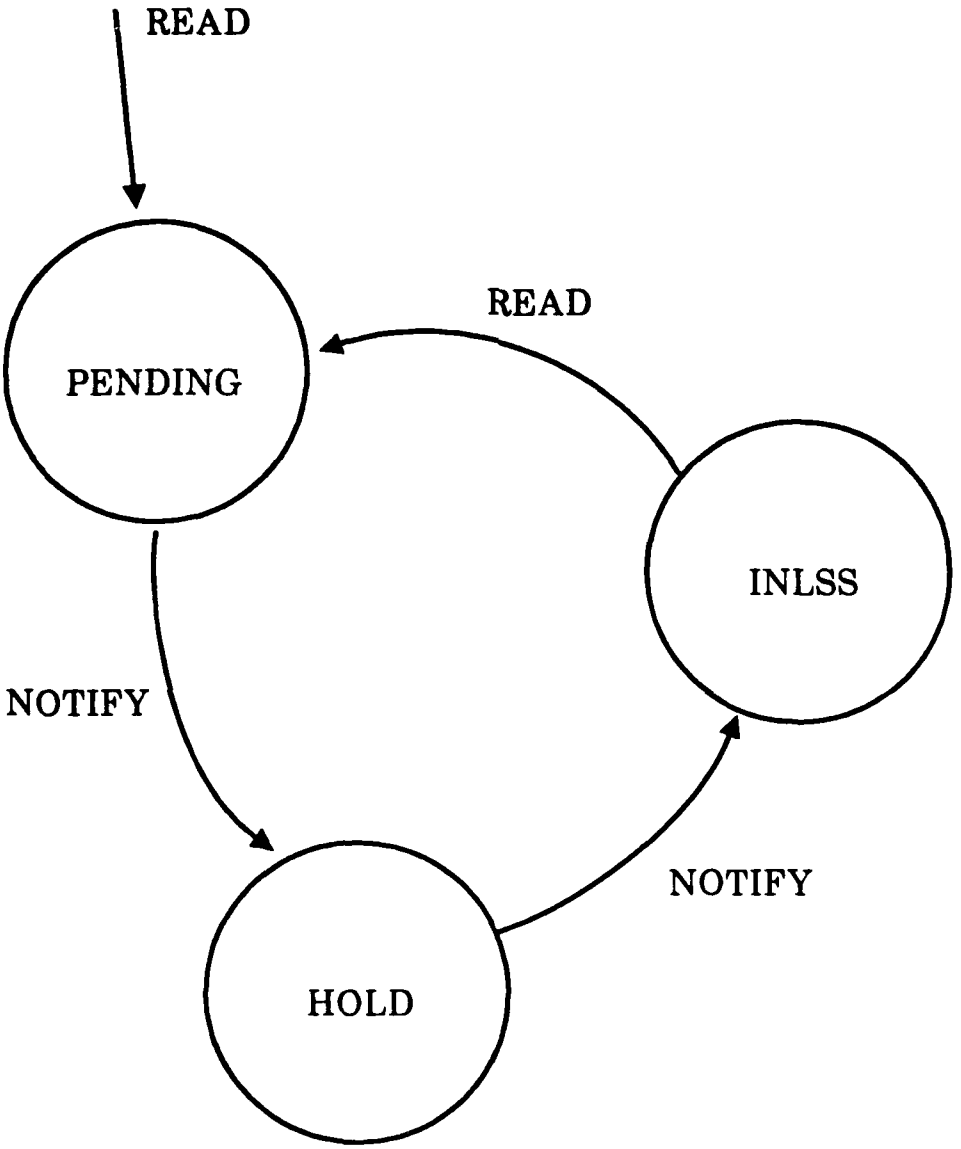


Figure 4.6 Status Transitions

In order for the GLOBAL-LRU-SOP algorithm to operate correctly, the class to class transitions must be restricted by certain logical constraints. Denote the number of slots in each class by m_f^i, m_s^i, and m_p^i, respectively. Then we have the following logical constraints:

L1: $m_s^i \leq m^i$, i.e., the number of data blocks stored at Level i can not exceed the capacity of Level i;

L2: No block, X, may be evicted unless $S^i(X) > m^{i-1}$, i.e., unless X's LRU stack position is strictly greater than the maximum capacity of Level i-1. This constraint represents the sufficiency conditions of Theorem 4, and guarantees that MLOI will hold;

L3: No block may be evicted unless its status is 'INLSS', i.e., unless the block is currently not being processed.

We can also propose operational constraints. These are constraints on the size of each class which are imposed for performance reasons. These constraints are defined in terms of parameters, p_j, which can be adjusted in order to tune the system. The operational constraints are:

O1: $m_s^i \geq p_1 m^i$, i.e., the utilization of storage at Level i should be as high as possible;

O2: $m_f^i \geq p_2$, i.e., the number of available free slots at Level i should not be allowed to sink below some predetermined number.

Appropriate values of p_j will be experimentally determined.

The EVICT process attempts to maintain O2 by moving slots from stored to free, without violating O1, if possible, and without violating L1, L2, or L3 under any circumstances. The logic of EVICT is as follows.

1) determine whether an EVICTION should be performed. If not, Exit. If so, find the slot number of the block to be evicted.

2) DELETE the slot to be evicted from the directory.

3) Put the evicted slot on the free slot chain.

4) Send a DELETE transaction to the LSS to free the LSS storage for the block.

4.3.8 Pipe-Lining

The algorithms presented in Section 4 derive a portion of their simplicity from the fact that they are "pipe-lined." By this we mean that the various processes that constitute a READ operation are designed to execute serially, rather than in parallel. In other words, each process terminates by generating the transaction which will awaken the next process to be executed. Opportunities for parallelism have been deliberately avoided. This strategy should not degrade performance, since in a multi-processor system, parallel processing of separate transactions is just as effective a way of utilizing the system resources as is parallel processing within individual transactions.

4.3.9 Multi-Processor Implementation Issues

The algorithms presented thus far assume a uni-processor environment. That is, no logic was included to lock shared data to prevent update inconsistencies. In order to run the algorithms in a multi-processor environment, they must be modified to lock shared tables during updates. This raises a number of issues:

- deadly embraces must be avoided,
- the most appropriate granularity for the locks must be determined, and

- the performance impact of contention for locked data must be estimated.

Presumably, deadly embraces will not be a serious problem. Since any process will need to have very few locks at any point in time, and will hold these locks for a very short period, it is reasonable to require processes to obtain all locks to be used for some function at the same time. This strategy will prevent deadly embraces, since functions will not start until their completion is assured.

The granularity issue is not so simple. Coarse granularity, i.e., locking entire tables, has the advantage of being simple and imposing little processing overhead. On the other hand, locking entire tables forces serial execution for functions that could perhaps be performed in parallel. Thus, system throughput is reduced. Fine granularity, i.e., locking individual table entries, allows as much parallel processing as is logically possible, thus minimizing contention for shared data. On the other hand, this strategy is more complex since a transaction may lock multiple entries, rather than a single table. Furthermore, more overhead is imposed in setting and resetting locks.

In order to estimate the performance impact of contention, suppose that some function requires 5 machine instructions which must be executed while some resource is locked. Using 1 MIP processors, this function could be performed at most 200,000 times per second, assuming that locking and unlocking the resource take no time at all. If the resource in question is the directory (using coarse locks), then the system is limited to at most 200,000 directory updates per second. If the locked resource is an individual directory entry, or a small set of entries (using fine grained locks), contention will be orders of magnitude lower than with coarse locks.

Thus, a fine grained lock strategy would seem to be necessary if DSH-III is to be able to process over 200,000 requests per second.

Unfortunately, the LRU_UPDATE function causes special problems. Recall that LRU updates must be performed in the same order at every level. The way this is achieved is by broadcasting LRU_UPDATE transactions, and then processing them at each level in the order in which they captured the GBUS. The requirement for FIFO processing effectively prevents parallel processing for these transactions. One possible way around this potential bottleneck is to divide the virtual address space into n segments, each with its own set of directory entries. Thus, each of the n segments would have an independent LRU chain. These chains could be updated in parallel, with FIFO processing within each segment. Assuming that references to virtual memory are evenly distributed among the n segments, this strategy would allow n-fold parallelism of LRU updates. The EVICT process could also operate in parallel. Therefore, the only interaction between segments would be the selection of the overall oldest slot from among the oldest slots on each of the n LRU chains. (Eviction must still be done in strict FIFO order, overall.) Note that this strategy implies that slots are time-stamped, as well as being chained together in LRU order.

The effect of this strategy is to replace contention due to updates by contention due to evictions. Assuming 90% locality, there are 10 times as many LRU updates as there are evictions at Level 1. Therefore, this strategy, by shifting contention from updates to evictions, could increase throughput by an order of magnitude.

The n-fold parallel eviction scheme could be implemented as follows. Reserve a n word area of shared memory, each word of which is to contain the time stamp of the oldest slot on the corresponding LRU chain. Dedicate a separate eviction process to each of the n LRU chains. Each of these processes would incorporate the following logic:

1) Find the oldest slot on LRU chain (each process is dedicated to a single LRU chain). Let tstamp = the time stamp of this slot.

2) Store tstamp in the appropriate location in the n word reserved area.

3) Compare tstamp with each of the other n-1 time stamps stored in the reserved area. If tstamp is not the smallest, repeat Step 3.

4) Evict the oldest slot on LRU chain, and release its LSS storage.

5) Go to Step 1.

Note that no locks are required to coordinate the n processes. All synchronization is provided by the shared list of time stamps which is updated by an atomic Store instruction in Step 2. Also note that the delay between evicting a block (in Step 4) and updating the time stamp list (in Step 2) causes no problems since the time stamps are monotonically increasing, and thus Step 3 always produces the correct result.

Finally, note that under the segmented address space strategy, the LRU update could use coarse locks, or their equivalent, since updates must be serialized within each segment of the address space. One way to accomplish this is to have n dedicated processes responsible for performing LRU updates for each of the n segments. Since there is only one process at a time updating any segment, this is effectively a uni-processor system, and no locks whatsoever are needed.

The avoidance of software bottlenecks in DSH-III by the use of strategies such as the one just suggested, will be a subject for further research.

5 ALGORITHMS SUPPORTING THE WRITE OPERATION

In Section 2, we discussed the advantages and disadvantages of four possible WRITE strategies. In this section, we present the software algorithms and transactions which can be used to implement one of these strategies, Staged Store Through.

5.1 Overview of the WRITE Operation

A WRITE request is initiated when a user sends a WRITE transaction to Level 1 of DSH-III. The format of the WRITE transaction is

WRITE(req_id,virt_addr,data)

where

req_id is a user assigned identifier for the request;

data is the N^0-byte block of data to be written (recall that N^0 is the size of the unit of transfer across the user interface of DSH-III);

virt_addr specifies the virtual address of data.

The first action taken by Level 1 is to make duplicate copies of data in separate memory modules. This minimizes the possibility of lost updates due to isolated recoverable memory failures at Level 1. In order to support this type of operation, we introduce a fifth LSS primitive, STORE_SPLIT. STORE_SPLIT takes an lss_id and a data block as arguments, and attempts to store the data block in a storage module which is physically independent of the memory module implied by lss_id. Thus, a sequence of LSS transactions and their responses to create duplicate copies of a update might be

```
STORE(virt_addr,data)                  - store first copy of data
STORE_ACK(virt_addr,lss_id1)           - receive lss_id from LSS
STORE_SPLIT(virt_addr,data,lss_id1) - store second copy
```

STORE_ACK(virt_addr,lss_id2) - receive second lss_id

After the data has been replicated, a WRITE_ACK message is sent to the user, informing him that the WRITE operation has been accepted by DSH-III. From the user's point of view, the WRITE has completed; all further processing for the WRITE request is handled internally to DSH-III.

Next, the N^1-byte page containing virt addr is modified. (A READ is issued to bring this page into Level 1 if it is not already there.)

Then the updated N^1-byte page is broadcast to all lower levels via a WRITE_BCAST transaction. Each level, j, upon receiving the broadcast, attempts to apply the update to the corresponding N^j-byte parent page of the updated page.

Finally, the duplicate copies of the original update are deleted from Level 1.

WRITE processing has a number of problems in common with the LRU updating discussed in Section 4. In particular, WRITEs must be performed in the correct sequence or else updates will be lost. Lam [15] addresses this problem in his Two Level Store Behind algorithm by associating a time-stamp with each page and with each of that page's sub-pages at the next higher level. Thus, each page in Level j will have $N^j/N^{j-1} + 1$ time-stamps associated with it. Using Lam's figures of $N^1 = 8$ and $N^2 = 128$, we have the result that 36% of the LSS storage at Level 2 is devoted to time-stamps, assuming that a time-stamp is 4 bytes in length. Unfortunately, the situation is even worse for Staged Store Through. The number of time-stamps associated with each page is one more than the size of the page divided by the size of the unit of update. Thus, for Staged Store Through, since the unit of update is N^1

bytes at every level, a page at Level j will have $N^j/N^1 + 1$ time-stamps. This implies that close to one third of the data base will be devoted to time-stamps.

The next section presents a set of WRITE algorithms which assume a uni-processor environment at each level. This allows us to serialize the processing of WRITEs, and thus guarantee that they are processed in the correct order. Section 5.3 presents a segmentation strategy, analogous to the one used to solve the LRU update problems, which allows n-fold overlapping of WRITE processing, while still ensuring that no updates are lost.

For this first version of the algorithm, we omit the details of making duplicate copies of the update in Level 1. These details are not an integral part of the logic of the WRITE operation.

5.2 WRITE Algorithms

This section presents the processing logic for each of the transactions used by the WRITE operation.

5.2.1 The WRITE Transaction - WRITE(req_id,virt_addr,data)

The WRITE transaction is sent by a user to Level 1 of DSH-III. The basic functions of the WRITE process are to enter the request onto the PRQ (creating a directory entry if necessary) and to retrieve the page to be updated. As for the READ operation, this retrieval may be either from the LSS or from lower down in the hierarchy. The logic of this transaction is almost identical to that for READ.

```
Select( SEARCH(virt_addr, lss_id, slot) );     /* search directory
When('NOTFOUND') Do;                           /* not in level 1
    DIR_ADD(virt_addr, slot);                  /* add to directory
```

```
    dir(slot).status = 'PENDING';                    /* note waiting
    PRQ_ADD(virt_addr, req_id, slot, ADDR(data));
                                                     /* add to PRQ
    READ(req_id, virt_addr, LEVEL+1);                /* retrieve page
    End /* of page not in level 1 */;
When('FOUND') Do;                                    /* page in level 1
    PRQ_ADD(virt_addr, req_id, slot, ADDR(data));
                                                     /* add to PRQ
    Select(dir(slot).status);                        /* branch on status
    When('HOLD');                                    /* already in LM
    When('PENDING');                                 /* already waiting
    When('INLSS') Do;                                /* page in LSS
        dir(slot).status = 'PENDING';                /* note waiting
        RETRIEVE(lss_id, slot, LSS(LEVEL));          /* retrieve from LSS
        End /* of page found in LSS */;
    Otherwise Error;                                 /* invalid status
    End /* of selection on status value */;
    End /* of page found in level 1 */;
Otherwise Error;                                     /* invalid SEARCH code
End /* of WRITE processing */;
```

5.2.2 The NOTIFY Transaction - NOTIFY(slot)

The logic of this process is very similar to the logic for the READ operation NOTIFY given in Section 4. The only difference is the incorporation of logic to handle WRITE requests on the PRQ, and the addition of the modified flag which the process uses to keep track of which blocks have been modified, and should therefore be rewritten into the LSS.

```
modified = False;                                    /* assume READs only
dir(slot).status = 'HOLD';                           /* lock page in LM

Do While( PRQ_POP(virt_addr, req_id, slot, ptr) ^= 'EMPTY' );
    Select( LEVEL );                                 /* branch on level
    When( 1 ) Do;                                    /* if level 1 ...
        Select(ptr);                                 /* READ or WRITE?
        When(NULL) Do;                               /* READ
            LRU_UPDATE(virt_addr, ALL);              /* send LRU update to all
                                                     /* levels
            DATA(dir(slot).lm_ptr, virt_addr, USER);
                                                     /* send data to user
            End /* of READ at level 1 */;
        Otherwise Do;                                /* WRITE
            MODIFY(dir(slot).lm_ptr, ptr, virt_addr);
                                                     /* apply update
            WRITE_BCAST(virt_addr, dir(slot).lm_ptr);
```

```
                                        /* broadcast update
            WRITE_ACK(req_id,USER);     /* signal completion
            modified = True;            /* note page modified
            End /* of level 1 WRITE processing */;
        End /* of level 1 processing */;
    Otherwise Do;                       /* if not level 1 ...
        Select(ptr);                    /* READ or WRITE?
        When(NULL) Do;                  /* READ
            DATA(dir(slot).lm_ptr, virt_addr, HIGHER(LEVEL));
                                        /* broadcast data to all
                                        /* higher levels
            End /* of READ at level j */;
        Otherwise Do;                   /* WRITE
            MODIFY(dir(slot).lm_ptr, ptr, virt_addr);
                                        /* apply update
            modified = True;            /* note page modified
            End /* of level j WRITE processing */;
        End /* of level j processing */;
    End /* of level selection */;
    RECYCLE;                            /* redispatch to enhance
                                        /* interleaving
End /* of loop over all requests on PRQ */;

If modified                             /* was page modified?
Then UPDATE(lm_ptr, slot);              /* yes, update LSS
Else Do;                                /* no, free LM page
    dir.status(slot) = 'INLSS';         /* note block is in LSS
    FREE(dir(slot).lm_ptr;              /* free block's LM storage
```

5.2.3 The UPDATE_ACK Transaction - UPDATE_ACK(slot)

This transaction is sent by an LSS to acknowledge the completion of an update to a previoiusly existing page of LSS storage.

```
dir(slot).status = 'INLSS';             /* note page in LSS
FREE(dir(slot).lm_ptr);                 /* release the LM storage
```

5.2.4 The WRITE_BCAST Transaction - WRITE_BCAST(virt_addr, data)

This transaction operationalizes the Staged Store Through operation, by broadcasting an updated N^1-byte page from Level 1 to all other levels of the hierarchy. Actually, the algorithm presented here represents Store Through, rather than Staged Store Through, since the broadcast of the update immediately follows the processing of the NOTIFY at Level 1. To implement a true Staged Store Through algorithm,

one would simply have to keep track of a list of queued update broadcasts, and actually do the broadcasts at convenient times e.g., just before the block overflows, or at a time of low GBUS utilization.

If a WRITE_BCAST does not find the block to be updated in a level, some error must have occurred. This is because READs and WRITEs are serialized in strict FIFO order in order to avoid lost updates. This implies that the WRITE_BCAST must be processed before any READs which left Level 1 at a later time. But these READs are the only possible way the page could have been forced out of Level 1, and therefore out of any lower level. This in turn implies that the WRITE_BCAST must be handled before the target page is evicted.

```
If SEARCH(virt_addr, lss_id, slot) = 'NOTFOUND' Then Error;
PRQ_ADD(virt_addr, dummy, slot, ADDR(data));
Select(dir(slot).status);
When('PENDING','HOLD');
When('INLSS') Do;
    RETRIEVE(lss_id, slot, LSS(LEVEL));
    End /* of retrieval of block for updating */;
End /* of status selection */;
```

5.2.5 The BCAST_DATA Transaction - BCAST_DATA(lm_addr, virt_addr)

The version of BCAST_DATA given in Section 4 discards the first PRQ entry at all levels except Level 1, since the broadcast has already sent the data to all higher levels. At first glance, it would appear necessary to distinguish between WRITE and READ PRQ entries, since WRITE entries must not be discarded. However, it turns out that the top PRQ entry can never be a WRITE entry when a BCAST_DATA is received. Therefore, the version of BCAST_DATA given in Section 4 will handle WRITES correctly.

5.3 Multi-Processor Implementation

The discussion of multi-processor implementation issues in Section 4 applies here as well. In particular, there are problems with order of execution of transactions in a multi-processor environment. In order to prevent "lost updates", WRITEs must be performed in the correct order, i.e., in the order in which they are presented to the system. Also, in order to produce consistent results for READs, WRITEs must be correctly sequenced with respect to READ operations. This situation is analogous to the LRU update problem, wherein LRU updates had to be performed in the correct order. The same solution is applicable.

1) Segment the address space and dedicate a process to each segment so that each N^1-byte page is always handled by the same process. This dedicated process serializes the READs and WRITEs for its segment of the address space. This implies that READ and WRITE handling must be combined into a single process at Level 1. Also, READ and WRITE_BCAST must be handled by a single process at all other levels.

2) Use 'S' message protocol for WRITE and WRITE_BCAST in order to assure correct ordering of WRITES with respect to READs. In essence, READs and WRITEs will be processed in FIFO order.

6 SUMMARY AND FURTHER RESEARCH

This paper has had three major goals. The goal of Section 2 was to present a coherent summary of the design issues and tradeoffs involved in the specification of an architecture for a Data Storage Hierarchy. While a lot of issues were raised and a lot of options discussed, most of the arguments pro and con in any area were based on experience with storage and file systems in general, and on intuition with regard to how a hierarchical system should behave under various conditions. There is very scanty relevant empirical data in this area. Performance evaluations of this type of structure [15, 29] have used macroscopic modeling methods such as simulation (e.g., GPSS), analytical queueing models [23], and Operational Analysis [7]. These efforts have provided valuable insights into the behavior of a hierarchical storage system under various macroscopic conditions (e.g., differing locality assumptions). However, these studies could not (because of the nature of their modeling methodologies) examine the reaction of the system to changes in microscopic architectural details, such as the choice of a replacement algorithm. The Software Test Vehicle (STV) [27], currently being built, will provide valuable insight into the effects of varying many of the detailed design parameters of DSH-III. These detailed simulation results could then be fed back as parameters into a GPSS or analytical model. This would allow examination of the macroscopic effects of microscopic design changes, thus answering many of the questions that were left open in Section 2 because of a lack of solid performance data.

Section 3 presented the design of a multi-processor implementation of DSH-III. Continuing work in this area will include designing a multi-processor operating system which can support the high throughput rates required by DSH-III.

Sections 4 and 5 presented a set of algorithms and transaction protocols which implement the READ and WRITE and automatic data migration functions of DSH-III. An STV implementation of these algorithms is planned. As noted above, the STV will allow investigation of various design questions. These include

- what is the best WRITE policy: Store Behind, Store Through, Store Replacement, or Staged Store Through?
- what is the best replacement policy? The basic algorithm GLOBAL-LRU-SOP will still work if LRU is replaced by any other replacement policy for which Theorem 4 holds. Perhaps there is a replacement policy which performs approximately as well as LRU, but which is much easier to implement.

This last point raises some interesting theoretical issues. It is conjectured that the eight theorems proved in [16] for LRU based policies can be extended to hold for a general class of replacement policies. The development of characterization theorems for this class of policies will immediately produce a means of identifying substitute replacement algorithms for DSH-III. A final interesting question is how this class of policies relates to the class of "stack" algorithms discussed in [20].

REFERENCES

[1]: Abdel-Hamid, T.K. and Madnick, S.E., 'A Study of the Multicache-Consistency Problem in Multi-Processor Computer Systems,' Proc. Sixth Workshop on Computer Architecture for Non-Numeric Processing, 1981.

[2]: Abe, Y., 'A Japanese On-Line Banking System,' Datamation, September 1977, pp 89-97.

[3]: Abraham, M.J., "Properties of Reference Algorithms for Multi-Level Storage Hierarchies,' Master's Thesis, Sloan School of Management, MIT, Cambridge, MA, June 1979.

[4]: Bartlett, J.F., 'A "NonStop" Operating System,' Tandem Computers Inc., Cupertino, CA, 1977.

[5]: Belady, L.A., 'A Study of Replacement Algorithms for a Virtual-Storage Computer,' IBM Systems Journal, Vol. 5, No. 2, 1966, pp 78-101.

[6]: Belady, L.A., Nelson, R.A. and Shedler, G.S., 'An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine,' Comm. ACM, Vol. 12, June 1969, pp 349-353.

[7]: Denning, P.J. and Buzen, J.P., 'The Operational Analysis of Queuing Models,' Computing Surveys, Vol. 10, No. 3, September 1978.

[8]: Dijkstra, E.W., 'The Structure of the "THE" Multiprogramming System,' Comm. ACM, Vol. 11, May 1968, pp 341-346.

[9]: Greenberg, B.S. and Webber, S.H., 'MULTICS Multilevel Paging Hierarchy,' IEEE Intercon, 1975.

[10]: Hemenway, J. and Grappel, R., 'Intel's iAPX "Micromainframe",' Mini-Micro Systems, May 1981, pp 73-89.

[11]: Hsu, M., 'A Preliminary Architectural Design for the Functional Hierarchy of the INFOPLEX Database Computer,' Working Paper No. WP1197-81, Sloan School of Management, MIT, Cambridge, MA, November 1980.

[12]: 3033 Processor Complex & 3033 Multiple Processor Complex Functional Characteristics, Form No. GA22-7060, International Business Machines Corp., White Plains, NY.

[13]: IBM 3850 Mass Storage System (MSS) Principles of Operation, Form No. GA32-0036, International Business Machines Corp., White Plains, NY.

[14]: System/370 Reference Summary, Form No. GX20-1850-3, International Business Machines Corp., White Plains, NY.

[15]: Lam, C., 'Data Storage Hierarchy Systems for Database Computers,' Doctoral Thesis, Sloan School of Management, MIT, Cambridge, MA, August 1979.

[16]: Lam, C. and Madnick, S.E., 'Properties of Storage Hierarchy Systems With Multiple Page Sizes and Redundant Data,' ACM Transcations on Database Systems, Vol. 4, No. 3, September 1979, pp 345-367.

[17]: Madnick, S.E., "Storage Hierarchy Systems,' Report No. TR-105, Project MAC, MIT, Cambridge, MA, 1973.

[18]: Madnick, S.E., 'Trends in Computers and Computing: The Information Utility,' Science, Vol. 185, March 1977, pp 1191-1199.

[19]: Madnick, S.E., 'The INFOPLEX Database Computer: Concepts and Directions,' Proc. IEEE Comp. Con., Februrary 1979, pp 168-176.

[20]: Mattson, R.L., Gecsei, J., Slutz, D.R. and Traiger. I.L., 'Evaluation Techniques for Storage Hierarchies,' IBM Systems Journal, Vol. 9, No. 2, 1970, pp 78-117.

[21]: Organick, E.I., The Multics System: An Examination of Its Structure, Cambridge, MA: MIT Press, 1972.

[22]: Ornstein, S.M., Crowther, W.R., Kraley, M.F., Bressler, R.D., Michel, A., and Heart, F.E., 'Pluribus - A Reliable Multiprocessor,' Proc. National Computer Conference, 1975, pp 551-559.

[23]: Reiser, M. and Kobayashi, H., 'Queuing Networks with Multiple Closed Queues: Theory and Computational Algorithms,' IBM Journal of Research & Development, Vol. 19, No. 3, May 1975.

[24]: Robidoux, S.L., 'A Closer Look at Database Access Patterns,' Master's Thesis, Sloan School of Management, MIT, Cambridge, MA, June 1979.

[25]: Rodriguez-Rosell, J., 'Empirical Data Reference Behavior in Data Base Systems,' Computer, November 1976, pp 9-13.

[26]: Simonson, W.E. and Alsbrooks, W.T., 'A DBMS for the U.S. Bureau of the Census,' Proc. Very Large Data Bases, September 1975, pp 496-497.

[27]: To, T., 'SHELL: A Simulator for the Software Test Vehicles of the INFOPLEX Database Computer,' Bachelor's Thesis, MIT, Cambridge, MA, June 1981.

[28]: Toong, H.D., Strommen, S.O. and Goodrich II, E.R., 'A General Multi-Microprocessor Interconnection Mechanism for Non-Numeric Processing,' Proc. Fifth Workshop on Computer Architecture for Non-Numeric Processing, 1980, pp 115-123.

[29]: Wang, R. 'Performance Evaluation of the INFOPLEX Data Base Computer,' Sloan School of Management, MIT, work in progress.

END

FILMED

5-86

DTIC